DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

PERCEPCIÓN DE LA ACTIVIDAD DE UN CONDUCTOR MEDIANTE CÁMARAS 3D

Trabajo Fin de Grado

Fecha: Mayo de 2013

Autor: Álvaro Loras Navas

Tutor: Dr. Arturo de la Escalera Hueso

“El verdadero progreso es el que pone la tecnología al alcance de todos”.

Henry Ford

INDICE GENERAL

INDICE DE FIGURAS

Agradecimientos.

Quisiera comenzar este Trabajo Fin de Grado mostrando mi agradecimiento a todos aquellos que, con su ayuda, lo han hecho posible.

En primer lugar a mi tutor, Arturo de la Escalera, por su interés y ayuda en cada etapa de realización de este proyecto, cuando no podía encontrar la manera de seguir adelante.

Agradecer a mi madre su apoyo y ánimos en todos los momentos difíciles de la carrera y de mi vida. Sin ella nunca podría haber llegado hasta aquí.

A mi novia, agradecerle la comprensión y paciencia por todas esas conversaciones sobre temas de ciencias, a veces tan difíciles de entender.

Y por último, quisiera agradecer a mi padre el estar siempre a mi lado y acompañarme en todo momento a lo largo de este juego, a veces tan injusto, que llamamos vida.

A todos vosotros, GRACIAS.

RESUMEN.

El objetivo que persigue este proyecto es la percepción de las actividades realizadas por un conductor de automóvil, mediante el sensor Kinect de Microsoft y el manejo de las librerías *PCL*(*Point Cloud Library*).

En primer lugar se introducirán las bases necesarias en la visualización 3D, para los usuarios que vayan a inicializarse en el uso del sensor Kinect.

Posteriormente se expondrán los algoritmos utilizados sobre la nube de puntos. Con los que se conseguirá segmentar, filtrar, esqueletizar y analizar los datos extraídos por el sensor.

Este proyecto muestra el gran potencial de las librerías *PCL*, tanto para los temas de los que trata este proyecto como para los relacionados con posibles trabajos futuros derivados de este proyecto y muchas otras aplicaciones de gran interés para la investigación en visión artificial.

Palabras clave:

*OpenCV, PC*L, visión artificial, segmentación, esqueletización, Kinect, *OpenNI*.

ABSTRACT.

The objective of this project is the perception of the activities done by the vehicle driver, captured by Microsoft's Kinect sensor and the use of PCL (Point Cloud Library).

In the first place, the necessary foundations will be introduced in the 3D visualization for the amateur Kinect users.

Later, the used algorithms will be exposed over the point cloud. With these algorithms, the data extracted by the sensor will be segmented, filtrated, skeletonized and analyzed.

This project shows the great potential of the PCL libraries for the topics of my project, but also for the future works related with it and many other applications of great interest for the artificial vision investigations.

Keywords:

OpenCV, PCL, artificial vision, segmentation, skeletal tracking, Kinect, OpenNI.

ABREVIATURAS.

CCD Charge **C**ouple **D**evice

CMOS Complementary **M**etal **O**xide **S**emiconductor

RGB Red **G**reen **B**lue

HSV Hue **S**aturation **V**alue

OpenCV Open Source **C**omputer **V**ision

SDK Software **D**evelopment **K**it

OpenNI Open Natural **I**nterface

PCL Point **C**loud **L**ibrary

Cmake Cross Platform **Make**

VB Visual **B**asic

NIR Near **I**nfra**r**ed

FPS Fotos **P**or **S**egundo

USB Universal **S**erial **B**us

SO Sistema **O**perativo

VTK Visualization ToolKit

GPU Graphics **P**rocessing **U**nit

CPU Central **P**rocess **U**nit

1. INTRODUCCIÓN.

A lo largo de este capítulo se expondrán los motivos y razones de la elección del proyecto. También se detallarán los objetivos a conseguir y se realizará una introducción a la visión por computador.

1.1. Motivación del proyecto.

Hoy en día, la visión por computador y sus aplicaciones experimentan un gran desarrollo tanto en tareas de la vida cotidiana como en investigación.

Esto se debe principalmente a que la visión por computador está ligada a la gran expansión tecnológica que vivimos: la aparición de los *Smart Phones, Tablets, Smart TV*… capaces de controlar varias de sus funciones con los gestos del usuario. También en el ámbito industrial o en biomedicina, la capacidad de controlar robots con movimientos corporales es hoy una realidad.

Por ello el 4 de noviembre de 2010, Microsoft lanzó al mercado de videojuegos la cámara Kinect para su videoconsola Xbox 360. Capaz de reconocer gestos, comandos de voz, y objetos e imágenes; sin necesidad de mandos ni cables.

La gran ventaja en su precio, tecnología y que el controlador sea de código abierto hacen que Kinect sea una herramienta fundamental en el ámbito de la investigación, lejos del mundo de los videojuegos.

1.2. Objetivos.

El objetivo principal de este proyecto será el desarrollo de una aplicación capaz de percibir la actividad del conductor de un vehículo mediante visión 3D.

Se filtrarán, segmentarán y analizarán los datos obtenidos del sensor, con el fin de obtener la esqueletización de los brazos del conductor y poder observar así las actividades que realiza dentro del vehículo.

Para ello se utilizará un ordenador personal, un sensor Kinect, la herramienta multiplataforma *Cmake*, las librerías "*Kinect for Windows SDK*", "*OpenNI*", "*PCL*" y "*OpenCV*". Previamente se realizará un análisis y comparación entre "*Kinect for Windows SDK*" y "*OpenNI*" para determinar cuál de las dos *frameworks* se usará en la resolución del proyecto.

1.3. Visión por computador.

La visión artificial o por computador intenta imitar la visión humana tratando de captar la información necesaria del mundo real a partir de imágenes para su posterior procesamiento en un computador. Dicha información abarca desde la detección de objetos hasta el análisis de imágenes tridimensionales.

Esta información visual proviene de energía luminosa, reflejada por los objetos del entorno y es captada por sensores sensibles a ella. En el caso del ser humano es el ojo el encargado de transformar esa energía en señales electroquímicas que posteriormente son enviadas al cerebro para su procesamiento. El ojo humano está formado por millones de pequeños receptores, llamados bastones y conos, sensibles a las variaciones luminosas.

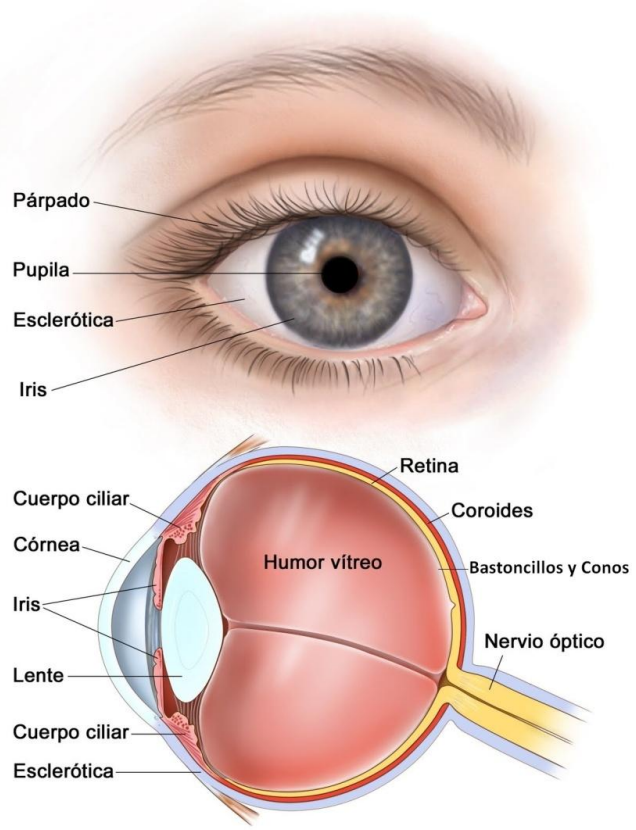


Figura 1. Partes del ojo humano. Disponible en: http://tescoleurs.blogspot.com.es/2010/05/percepcion-de-la-luz-entre-el-ojo-y-el.html

El dispositivo encargado de sustituir al ojo humano en la visión por computador es una cámara digital, la cual consigue transformar la información luminosa en señales digitales capaces de ser procesadas por un computador. También existen otros tipos de cámaras (las analógicas) que hoy en día están cada vez más en desuso, ya que las cámaras digitales no necesitan un pre-proceso de digitalización que permita al computador procesar los datos de la imagen.

Las cámaras digitales son capaces de capturar la información gracias a las matrices de receptores instaladas en su interior. Cada elemento de la matriz se denomina píxel y pueden ser de diferentes tecnologías (*CCD o CMOS*). En el caso de querer obtener la información en escala de grises, necesitaremos un receptor por cada píxel y 3 si deseamos la imagen en color.

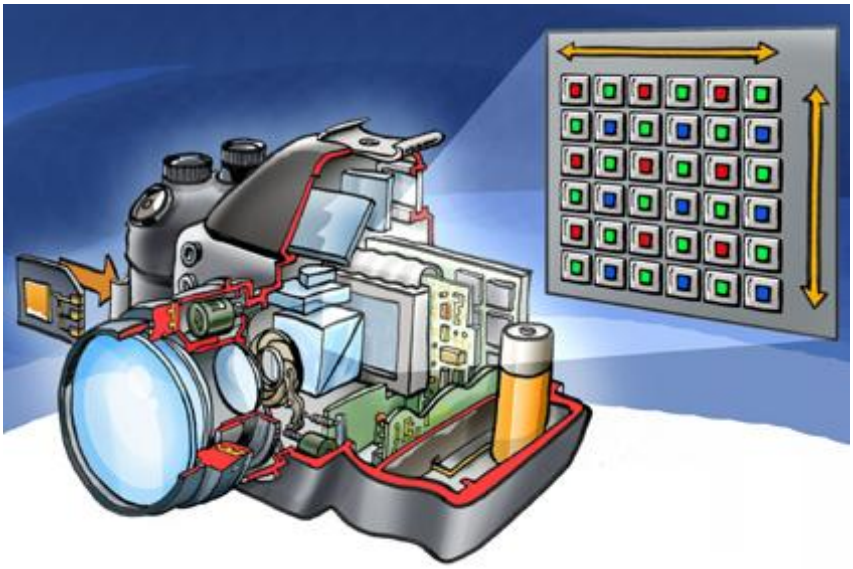


Figura 2. Cámara digital. Disponible en:
http://es.wikipedia.org/wiki/C%C3%A1mara_digital

Lo explicado hasta el momento hace referencia a imágenes bidimensionales, pero… ¿Cómo se puede extrapolar este conocimiento a las 3 dimensiones?

Para obtener el equivalente en 3D se trabaja con nubes de puntos.

Podemos definir una nube de puntos como una matriz tridimensional cuya mínima expresión es un punto (véase la diferencia con el píxel), cada punto puede contener información relativa al color, ya sea RGB (*Red Green Blue*), HSV (*Hue Saturation Value*) o escala de grises y las coordenadas del punto representan su posición en el entorno real en X, Y y Z.

En nuestro caso se obtendrá una nube de puntos que representa, mediante las librerías de PCL, los datos que se obtienen a través de nuestro sensor de infrarrojos y de la cámara RGB, de tal forma que cada uno de estos datos tiene integrados datos relativos a la localización espacial de coordenadas X, Y y Z, entendiendo que el eje de coordenadas es la propia cámara, y de color, ya sea RGB, HSV o escala de grises. Estos datos se visualizan en un entorno que utiliza internamente VTK (*Visualization ToolKit*) y OpenGL para la interacción mediante eventos con dichos datos por parte del usuario.

En la figura 3 se puede observar la nube de puntos que representa un cilindro en 3D.

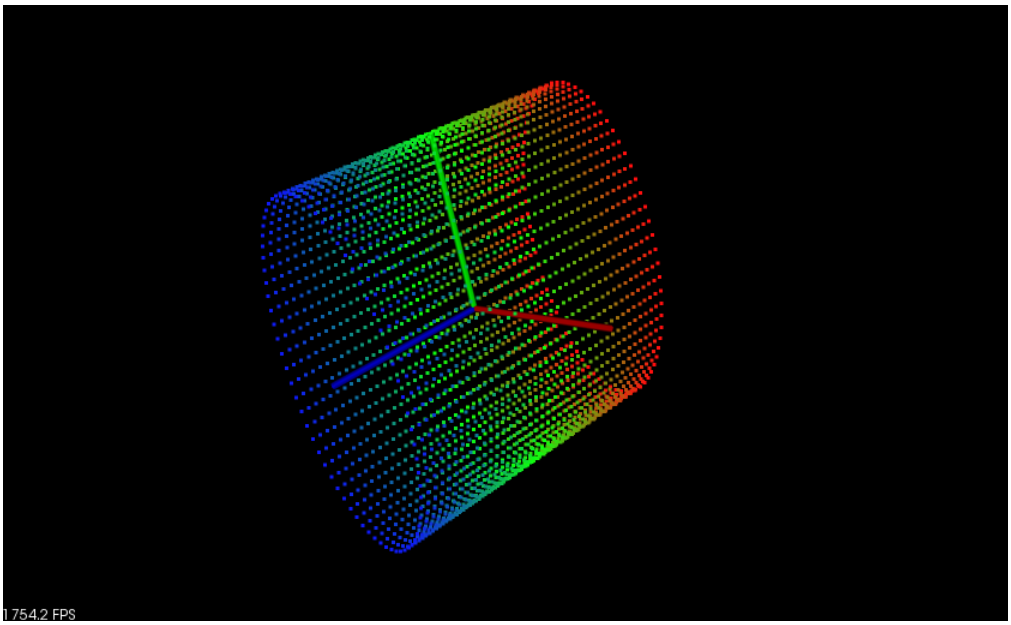


Figura 3. Nube de puntos de un cilindro. Disponible en:
http://pointclouds.org/documentation/tutorials/_images/pcl_visualizer_color_rgb.png

1. INTRODUCTION.

Across this chapter, there will be exposed the reasons of the project, also the achieving aims will be detailed and it will be done an introduction to the artificial vision.

1.1. Project's motivation.

Nowadays, the artificial vision and its applications experience a great development in every life tasks as well as in investigation.

This is because the artificial vision is linked to the great technological expansion that we are living: the emergence of Smart Phones, Tablets, Smart TV… able to control many of the functions with the user's gestures. It is also a fact the capacity of controlling robot with corporal movements in the industrial ambit or in biomedicine.

Thus, on 4th November 2010, Microsoft brought out the Kinect camera to the videogames market for its videogame console Xbox 360, which is able to recognize gestures, voice commands, objects and images without the necessity of controls and cables.

The great advantage of its price, its technology and that the controller is of an open code, convert Kinect in a basic tool in the investigation ambit, far away from the videogames world.

1.2. Objetives.

The principal objective of this project will be the development of an application able to perceive the driver's activity in a vehicle through 3D vision.

It will be filtered, segmented and analyzed all the achieved data by the sensor with the aim to obtain the skeletal tracking of the driver arms and to observe the activities that he realize inside of the vehicle.

To do this it will be use the personal computer, a Kinect sensor, the Cmake multi-platform tool, the libraries "Kinect for Windows SDK", "OpenNI", "PCL" and "OpenCV". Then it will be realized an analysis and the comparison between "Kinect for Windows SDK" and "OpenNI" to determine which of the both frameworks will be used for the project resolution.

1.3. Computer Vision.

The artificial vision or by computer tries to imitate the human vision to capture the necessary information of the real world from images for its further processing in a computer. That information includes from detection of objects to the analysis of three-dimensional images.

This visual information comes from the light energy reflected by the around objects and it is captured by sensitive sensors. In the case of human being is the eye which transforms this energy in electrochemical signals that would be later sent to the brain for its processing. The human eye is formed by millions of small receptors, called rods and cones, sensible to the light variations.

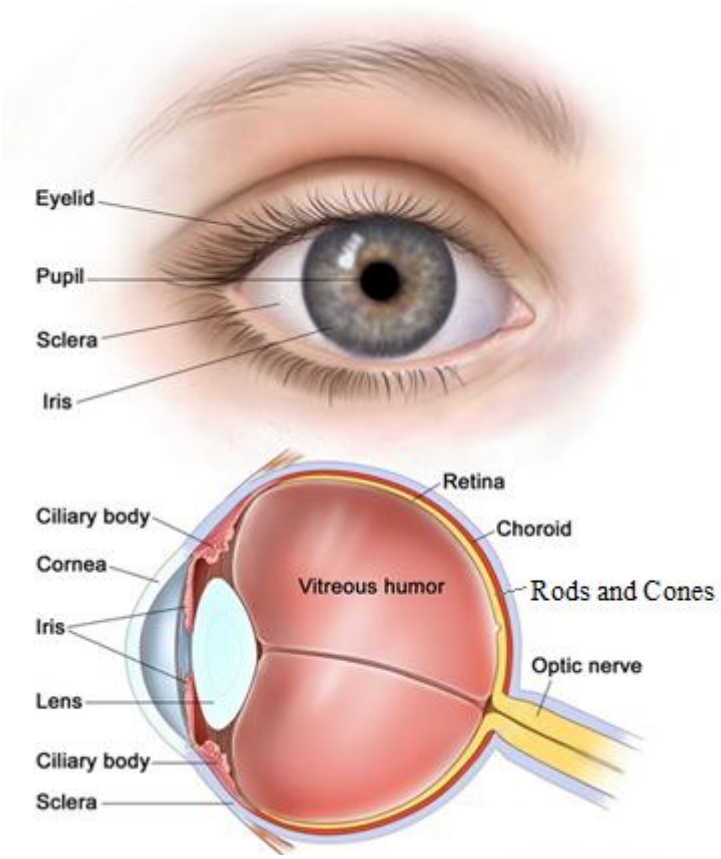


Figure 4. Parts of human eye. Available in:
http://cdn3.dogonews.com/pictures/9268/content/Eye%20Diagram.jpg?1322084390

The device in charge of replacing the human eye in the artificial vision is a digital camera which transforms the light information in digital signals able to be processed by a computer. Also exist other types of camera (analogical) that nowadays are less used. Because the digital cameras don't need a pre-process of digitalization which allows the computed to process the image information.

The digital cameras are capable to capture the information thanks to the matrixes of the receptors installed in its interior. Each element of this matrix is called pixel and it can belong to different technologies (CCD or CMOS). In case you want to obtain the information in grayscale, we will need a receptor for each pixel and 3 if we want a color image.

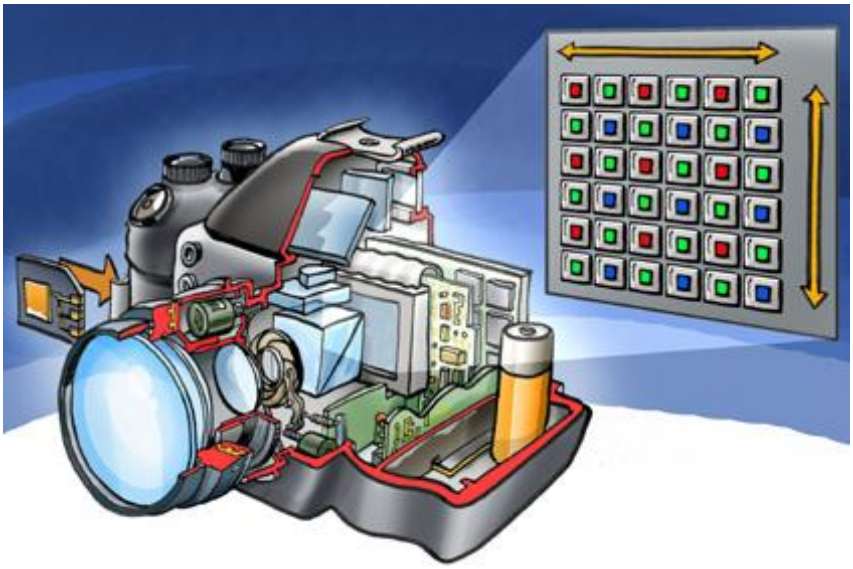


Figure 5. Digital camera. Available in:
http://es.wikipedia.org/wiki/C%C3%A1mara_digital

What has been explained so far makes reference to two-dimensional images, but… How can this knowledge be extrapolated to the 3 dimensions?

To achieve the equivalent in 3D we work with point clouds.

Point clouds can be defined as a three-dimensional matrix whose minimal expression is a point (take a look to the difference with a pixel), each point can contain relative information to the color, it can be RGB, HSV or grayscale and the point coordinates represent their position in the real setting in X,Y and Z.

In our case, it will be obtained a point cloud which represents the achieved data through PCL libraries and through our infrared sensor and the RGB camera in such a way that this information has integrated relative data to the spatial localization of the coordinates X, Y and Z, understanding that the coordinate axis is the own camera, of color, which can be RGB, HSV, or grayscale. This information is displayed in a setting that uses internally VTK and OpenGL for the interaction through events by the user with this data.

In the figure 3, it can be observed a point cloud which represents a cylinder in 3D.

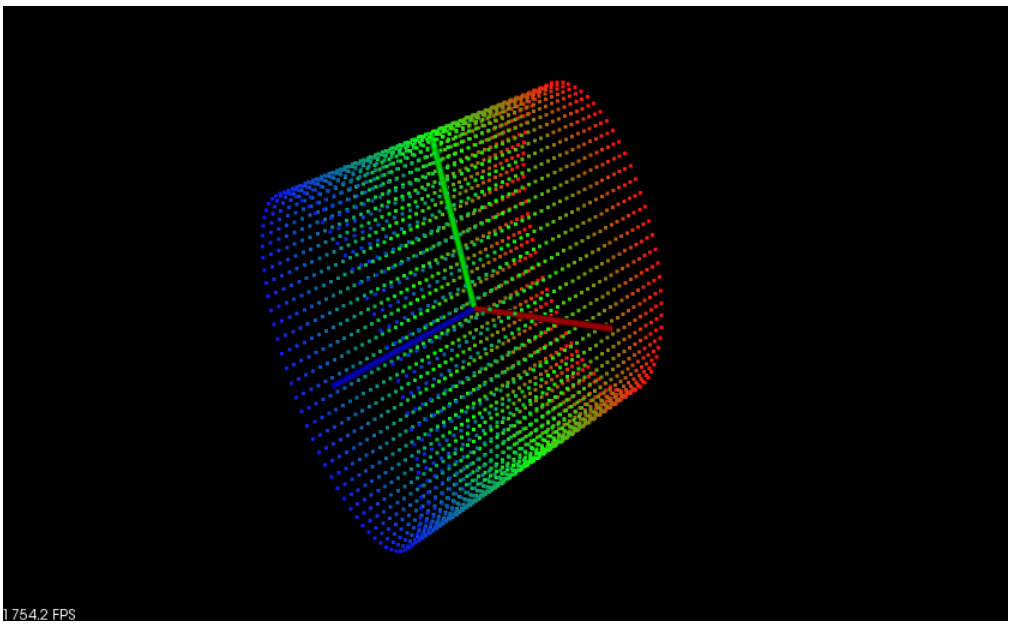


Figure 6. Point cloud of a cylinder. Available in:
http://pointclouds.org/documentation/tutorials/_images/pcl_visualizer_color_rgb.png

2. CARACTERÍSTICA DE LA CÁMARA KINECT.

El controlador de juego libre y entretenimiento Kinect, originalmente llamado "Proyecto Natal", fue creado por Alex Kipman y desarrollado por Microsoft para la videoconsola Xbox 360. Posteriormente se saco un nuevo modelo para PC a través de Windows 7 y Windows 8.

Kinect permite a los usuarios controlar e interactuar con la videoconsola sin necesidad de controladores físicos. A través de su interfaz reconoce gestos, comandos de voz, y objetos e imágenes.

2.1. Especificaciones técnicas.

El dispositivo Kinect dispone de un Chip PrimeSense PS1080-A2 para procesamiento de los mapas, una cámara RGB (*Red, Green, Blue*), una cámara NIR (*Near Infrared*) y un emisor de luz infrarroja que permite obtener datos de profundidad por la cámara NIR, incluso en ausencia de luz.

En la siguiente figura se puede observar la cámara RGB corresponde a la cámara central del dispositivo, la NIR se encuentra a la derecha y el emisor de luz infrarroja a la izquierda. Una vez conectamos Kinect es fácil distinguir el emisor de luz infrarroja.

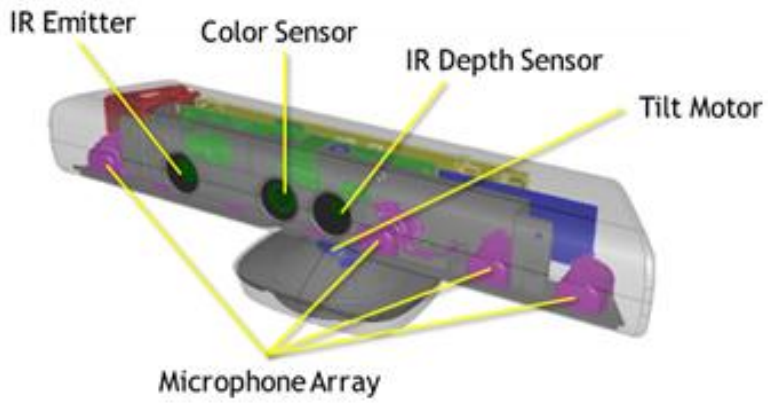


Figura 7. Partes de Kinect. Disponible en:
http://malenyabrego.wordpress.com/category/kinect/

Además Kinect dispone de un motor capaz de mover el dispositivo verticalmente y 4 micrófonos capaces de ubicar la fuente del sonido y filtrar el ruido ambiente.

A continuación se mostrará una tabla comparativa entre los dos modelos existentes de la cámara Kinect.

Tabla comparativa especificaciones Kinect Xbox 360 y Kinect for Windows.

Información disponible en:
http://www.kinectfordevelopers.com/es/2012/03/01/diferencias-entre-kinect-xbox-360-y-kinect-for-windows/

Sensores

Kinect for XBOX 360	Kinect for Windows
Lentes de color y sensación de profundidad	*[Se presupone que igual o mejor]*
Micrófono multi-arreglo	*[Se presupone que igual o mejor]*
Ajuste de sensor con su motor de inclinación	*[Se presupone que igual o mejor]*
Totalmente compatible con las consolas existentes de Xbox 360. Compatible con PC mediante el uso de un cable USB que se ha de comprar a parte.	Pensado para trabajar en PC bajo Windows 7, Windows Embedded Santard 7 y Windows 8.
[¡Novedad!]	Se ha mejorado la robustez de los sensores, entre ellos la estabilidad de los drivers, el tiempo de ejecución y se han hecho varios arreglos en el audio.

Data Streams (Flujo de datos)

Kinect for XBOX 360	Kinect for Windows
320 × 240 a 16 bits de profundidad @ 30fps	80 x 60, 320 × 240, 640 x 480 @ 30fps
640 × 480 32-bit de color @30fps	**640 × 480 32-bit de color @30fps** **1280 x 960 RGB @ 12fps** **Raw YUV 640 x 480 @ 15fps**
Audio de 16-bit @ 16 kHz	Audio de 32 y 64 bits, en función del SO.

Campo de visión

Kinect for XBOX 360	Kinect for Windows
Campo de visión horizontal: 57 grados	*[Se presupone que igual o mejor]*
Campo de visión vertical: 43 grados	*[Se presupone que igual o mejor]*
Rango de inclinación física: ± 27 grados	*[Se presupone que igual o mejor]*
Rango de profundidad del sensor: 1,2 – 3,5 metros	**Default Mode** • 0 – 0,8 metros, fuera de rango • 0,8 – 4 metros, parámetros normales • 4 – 8 metros, se recoge información pero no es óptima. • > 8 metros, fuera de rango **Near Mode** • 0 – 0,4 metros, fuera de rango • 0,4 – 3 metros, parámetros normales (la mejor calidad se encuentra a los 2 metros). • 3 – 8 metros, se recoge información pero no es óptima. • > 8 metros, fuera de rango

***NOTA** – Microsoft asegura que el sensor físicamente es el mismo, para Kinect de XBOX 360 ya se podían captar esqueletos situados a 50 centímetros de distancia, bajo la nota de "siempre que haya una buena iluminación". Solo se ha necesitado cambiar el firmware.*

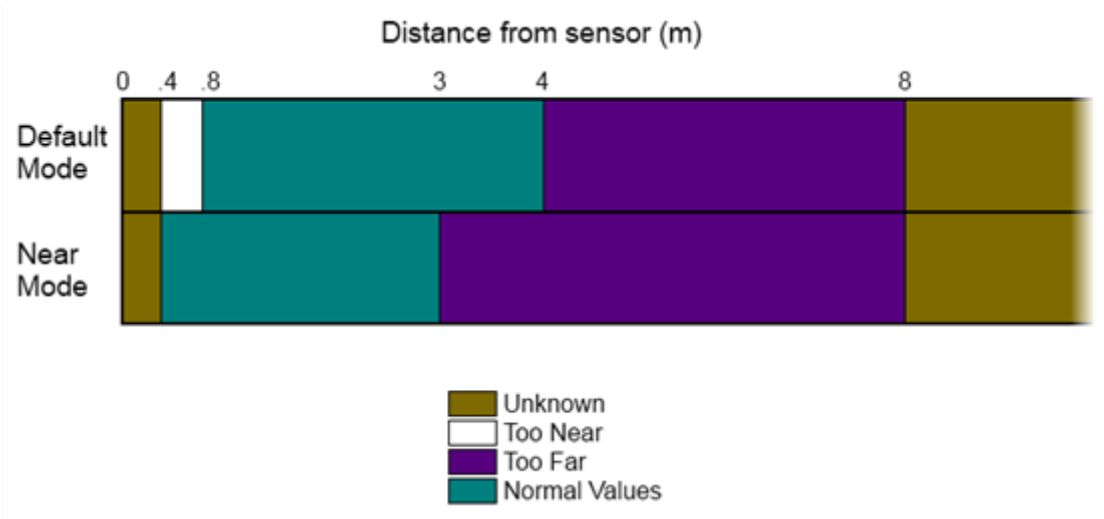


Figura 8. Modos Rango de Visión. Disponible en: **http://www.kinectfordevelopers.com/es/2012/03/01/diferencias-entre-kinect-xbox-360-y-kinect-for-windows/**

Sistema de Seguimiento

Kinect for XBOX 360	Kinect for Windows
Rastrea hasta 6 personas, incluyendo 2 jugadores activos	Sigue igual, esperan alcanzar los 6 jugadores en futuras versiones.
Rastrea 20 articulaciones por jugador activo	• **Default Mode** - 20 articulaciones. • **Near Mode** – No aseguran la cantidad, pero si que en futuras versiones del software esperan conseguir la misma capacidad que en el Default Mode.
Permite utilizar un único sensor por equipo	Permite utilizar hasta 4 sensores a la vez en el mismo equipo bajo las siguientes premisas: • Cada sensor debe estar en un puerto USB diferente. • El PC debe tener potencia suficiente para poder soportar los 4 sensores.
[¡Novedad!]	**Skeletal Tracking -** Se ha mejorado el rastreo y seguimiento de esqueletos como se hizo en la "Skeletal Tracking Library" que se lanzó para el sensor de Xbox 360 en Noviembre de 2011.

Sistema de audio

Kinect for XBOX 360	Kinect for Windows
Reconocimiento de voz múltiple	Ahora permite el reconocimiento de hasta 26 lenguas distintas en el “Microsoft Speech Platform Runtime Languages v11.0”
[¡Novedad!]	Se han realizado bastantes mejoras en todo lo relacionado con el audio del sensor debido a las deficiencias detectadas en el dispositivo des de su comercialización hace algo más de un año.

2.2. Método de obtención de la profundidad.

La tecnología PrimeSense utilizada por Kinect presenta un método de adquisición de datos el cual proyecta un patrón, pseudo aleatorio, de puntos infrarrojos conocido. Una vez obtenido se mide la deformación de la malla de puntos proyectada. Es decir, la distorsión del patrón en cuanto a lo que se refiere a la profundidad de los puntos que lo componen.

Figura 9. Malla de puntos. Disponible en:
http://www.elmundo.es/blogs/elmundo/el-gadgetoblog/2010/11/08/asi-ve-kinect-tu-salon.html

El sistema de PrimeSense presenta una codificación única y directa, con el fin de que cada posición sea reconocida para cada punto del patrón y elimina la mayor parte del ruido producido por los rayos infrarrojos presentes en la luz solar. Aunque también hay que decir, que este dispositivo presenta serias dificultades de funcionamiento en zonas abiertas, debido a la gran concentración de estos rayos.

En la figura 7 se observa el sistema de funcionamiento presentes en los sensores PrimeSense.

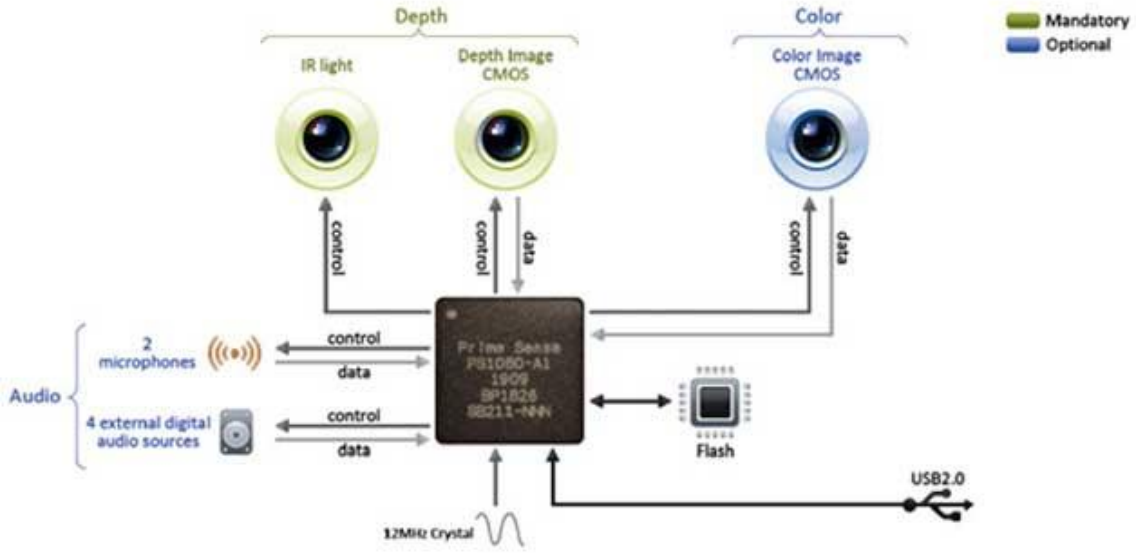


Figura 10. Controlador PrimeSense. Disponible en:
http://www.laquees.com/2010/11/no-desarmes-tu-kinect/

Para calcular los datos de profundidad, se siguen los siguientes pasos:

1. **El sensor PrimeSense se calibra en el momento de su fabricación.**
2. **Se calcula la triangulación entre la malla de puntos infrarrojos distorsionada y el modelo del patrón.**
3. **Cada uno de los puntos tienes su correspondencia en el patrón.**
4. **Se obtiene el mapa de profundidad.**

3. HERRAMIENTAS UTILIZADAS.

3.1. Sistema Operativo.

Este proyecto se ha desarrollado íntegramente en Windows 7 de 64 bits. El principal motivo de elegir este entorno es que Windows sigue siendo el SO más utilizado en todo el mundo hasta la fecha. También porque es compatible con todo el resto de herramientas y aplicaciones necesarias para cumplir las especificaciones de este proyecto.

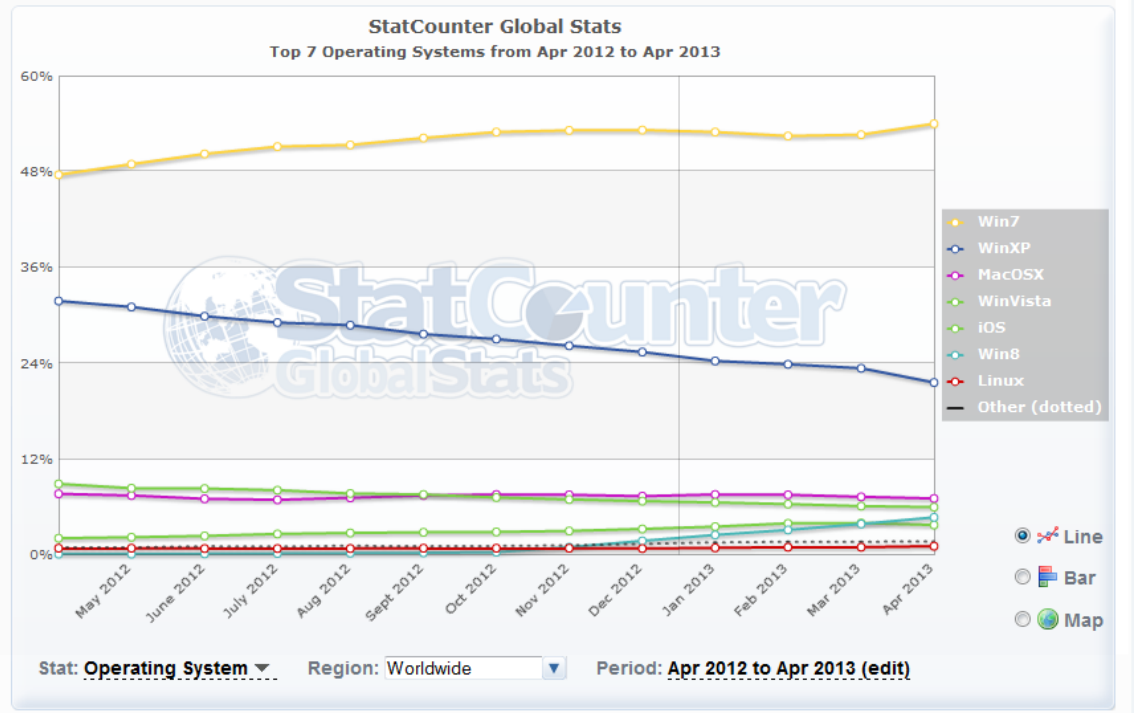


Figura 11. Estadística uso de SO. Disponible en:
http://gs.statcounter.com/#os-ww-monthly-201204-201304

Windows 7 es una versión de Microsoft Windows, línea de sistemas operativos producida por Microsoft Corporation. Esta versión está diseñada para uso en PC, incluyendo equipos de escritorio en hogares y oficinas, equipos portátiles, tablet PC, netbooks y equipos media center.

El desarrollo de Windows 7 se completó el 22 de julio de 2009, siendo entonces confirmada su fecha de venta oficial para el 22 de octubre de 2009 junto a su equivalente para servidores Windows Server 2008 R2.

3.2. Lenguaje C++.

El lenguaje de programación utilizado en este proyecto es el C++ ya que es un lenguaje intuitivo y fácil de programar. Además, todas las herramientas utilizadas en la realización de este proyecto se pueden programar en este lenguaje, tanto OpenNi, Windows SDK y PCL.

El origen del lenguaje de nivel superior C++ es el lenguaje de nivel medio C. El lenguaje C se creó en 1972 por los Laboratorios Bell orientado a la implementación de Sistemas Operativos, concretamente Unix. Su eficiencia y facilidad en la ceración de código le han convirtieron en el lenguaje de programación más popular para crear software de sistemas, aunque también se utiliza para crear aplicaciones.

En 1980 el lenguaje C evolucionó al denominado C con Clases, adquiriendo nuevas funcionalidades como argumentos de funciones, clases, etc… y más tarde, en 1983 este leguaje fue rediseñado. Obteniendo la capacidad de implementar funciones virtuales, funciones sobrecargadas y operadores sobrecargados. A esta nueva versión se la denominó C++, convirtiéndose en el lenguaje de nivel superior que conocemos hoy en día.

Otras características del Lenguaje C++:

- Compatibilidad con el Lenguaje C.
- Orientación a objetos.
- Programación modular.
- Agrupación de instrucciones.
- Concepto de puntero.
- Los argumentos de las funciones se transfieren por su valor.

3.3. OpenCV.

OpenCV es una librería libre de visión artificial para el tratamiento de imágenes. Originalmente fue desarrollada por Intel Corporation en 1999 y gracias a que fue publicada bajo licencia BSD, se pudo utilizar en multitud de aplicaciones con fines comerciales y de investigación: como sistemas de seguridad con detección de movimiento o el control de procesos mediante reconocimiento de objetos.

Es un software multiplataforma que utiliza el Lenguaje C y C++ y se puede utilizar tanto en aplicaciones para GNU/Linux, MacOS y Microsoft.

La principal característica de OpenCV es la funcionalidad, claridad y sencillez de uso de sus funciones. Los algoritmos están basados en estructuras de datos.

Dispone de más de 500 funciones dedicadas a:

- Transformaciones geométricas.
- Imágenes Piramidales.
- Procesamiento general de imágenes.
- Segmentación.
- Descriptores.
- Calibración de cámaras.
- Detección de rostros.

OpenCV se pude descargar en (sourceforge.net/projects/opencvlibrary)

3.4. Point Cloud Library (PCL).

La librería Point Cloud Library (PCL) es un macro código abierto independiente que incluye numerosas técnicas para el análisis y procesamiento de nubes de puntos en 3D. Esta librería contiene una gran cantidad de algoritmos para filtrar, modelar, segmentar, reconstruir superficies, mapeado y reconocimiento de objetos, etc.

A continuación se muestra el conjunto más importante de módulos liberados en PCL.

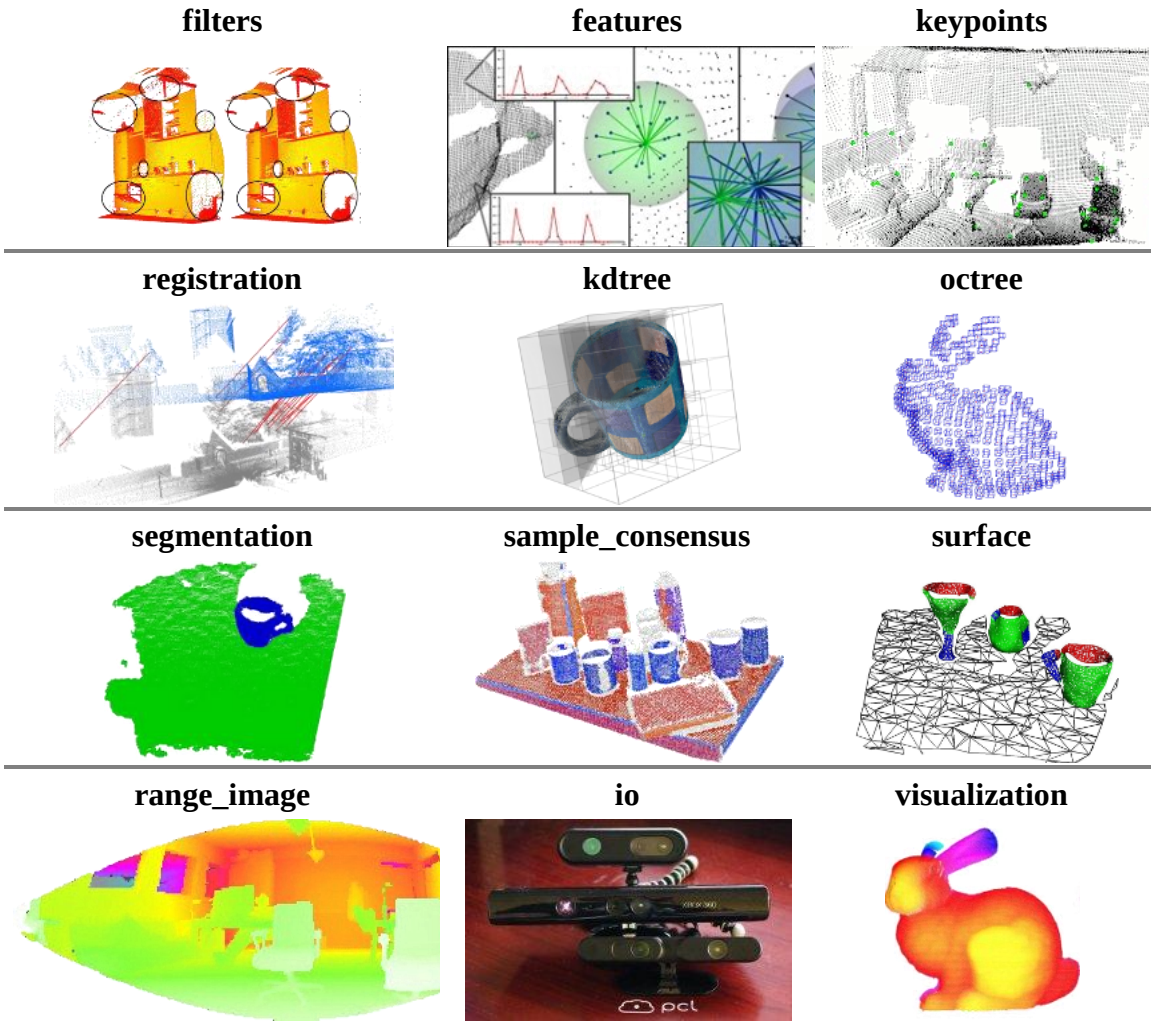


Figura 12. Librerías de PCL. Disponible en:
http://pointclouds.org/documentation/

Filters.

Un ejemplo de filtro que se puede encontrar en PCL es el de supresión de ruido. Debido a errores de medición, determinados conjuntos de datos presentan un gran número de puntos de sombra o ruido que no deberían aparecer. Este factor complica la estimación de los puntos 3D característicos locales.

Algunos de estos valores extremos se pueden filtrar mediante la realización de un análisis estadístico en el vecindario de cada punto, y el recorte de los que no cumplen con ciertos criterios.

El algoritmo **Statistical Outlier Removal** se basa en el cálculo de la distribución de la distancia a puntos vecinos del conjunto de los datos de entrada. Para cada punto, se calcula la distancia media desde él a todos sus vecinos. Suponiendo que la distribución resultante es gaussiana con una media y una desviación estándar, todos los puntos cuya distancia media están fuera de un intervalo definido por las distancias globales medias y la desviación estándar, se pueden considerar como valores atípicos y se recortan del resto de datos.

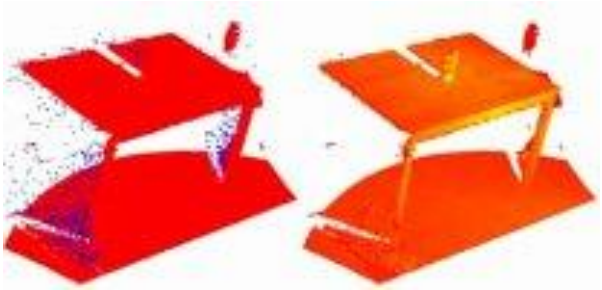


Figura 13. Statistical Outlier Removal de PCL. Disponible en:
http://pointclouds.org/documentation/tutorials/index.php

Otro tipo de filtro es **Pass Through**. Con este método, se consigue filtrar los datos de la nube de puntos en función de la profundidad de las tres dimensiones X, Y y Z. Se puede conseguir introduciendo como parámetros los intervalos con los que nos queremos quedar en la nube de puntos y desechar el resto.

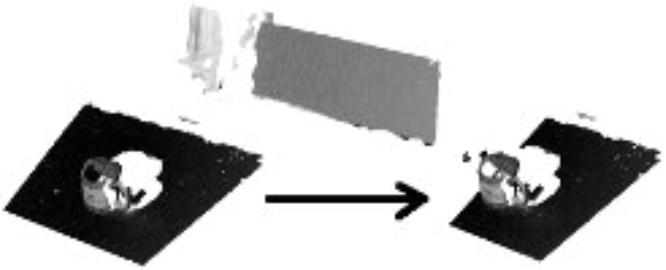


Figura 14. Pass Through de PCL. Disponible en:
http://pointclouds.org/documentation/tutorials/index.php

Features.

La librería Features contiene funciones 3D para la estimación de los datos de las nubes de puntos. Estas funciones 3D son representaciones o posiciones en el espacio, que describen los patrones geométricos basados en la información disponible en torno al punto.

El espacio de los datos seleccionados alrededor del punto de consulta se refiere generalmente como el barrio-k.

Dos de las características geométricas de puntos más utilizadas son la curvatura estimada de la superficie y la normal a un punto p de consulta. Ambas se consideran características locales, ya que caracterizan a un punto utilizando la información proporcionada por sus k vecinos más inmediatos. Para la determinación de estos vecinos de manera eficiente, el conjunto de datos de entrada está generalmente dividido en partes más pequeñas, utilizando técnicas de descomposición espacial como octrees o KD-árboles, y las búsquedas de puntos más cercanos.

Dependiendo de la aplicación puede optarse por determinar cualquier número de puntos fijos k en la vecindad de p, o todos los puntos que se encuentran dentro de una esfera de radio r con centro en p. Uno de los métodos más sencillos para la estimación de las normales de la superficie y los cambios de curvatura en un punto p es realizar una eigendecomposition (es decir, calcular los vectores propios y valores propios) de la superficie de puntos barrio-k.

Keypoints.

Esta librería contiene implementaciones de dos algoritmos de detección de Keypoints (puntos clave) de una nube de puntos.

Keypoints (también denominado como puntos de interés) son puntos en una imagen que son estables, distintivo, y se pueden identificar usando un criterio de detección bien definido. Típicamente, el número de puntos de interés en una nube de puntos será mucho menor que el número total de puntos en la nube, y cuando se utiliza en combinación con la característica de descriptores locales en cada punto significativo, los puntos clave y descriptores puede ser utilizado para formar una compacta, pero descriptiva, representación de los datos originales.

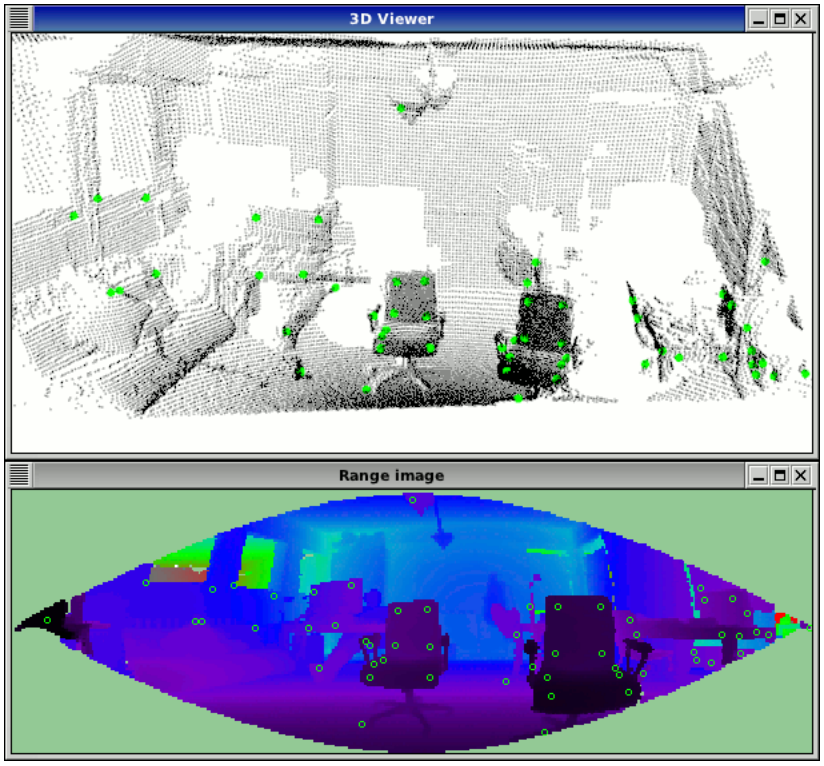


Figura 15. Keypoints de PCL. Disponible en:
http://pointclouds.org/documentation/tutorials/narf_keypoint_extraction.php#narf-keypoint-extraction

Registration.

La combinación de varios conjuntos de datos en un modelo coherente mundial se realiza por lo general mediante una técnica denominada registro. La idea clave es identificar los puntos correspondientes entre los conjuntos de datos y encontrar una transformación que minimiza la distancia (error de alineación) entre los puntos correspondientes. Este proceso se repite, desde que la búsqueda correspondiente se ve afectada por la posición relativa y la orientación de los conjuntos de datos.

Una vez que los errores de alineación caen por debajo de un determinado umbral, el registro se dice que es completo. La librería de registro implementa una gran cantidad de algoritmos de registro de la nube puntos para ambos conjuntos de datos organizados y no organizados (propósito general). Por ejemplo, PCL contiene un conjunto de algoritmos potentes que permiten la estimación de varios conjuntos de correspondencias, así como los métodos para rechazar malas correspondencias, y la estimación de las transformaciones de una manera robusta.

KdTree.

La librería kdtree PCL proporciona el kd-árbol de estructura de datos, utilizando FLANN, que permite búsquedas rápidas de vecinos más cercanos.

Kd-tree (árbol k-dimensional) es una estructura de datos espaciales que almacena un conjunto de puntos k-dimensionales en una estructura de árbol que permite búsquedas por rango eficientes y búsquedas de vecinos más cercanos.

Búsqueda de vecinos más cercanos son una operación principal cuando se trabaja con datos de nubes de puntos y se puede utilizar para encontrar correspondencias entre los grupos de puntos o características o descriptores para definir el entorno local de uno o varios puntos.

Octree.

La librería octree proporciona métodos eficientes para la creación de una estructura de datos de árbol jerárquico de los datos de la nube de puntos. Esto permite particionamiento espacial, disminución de la resolución y operaciones de búsqueda en el conjunto de datos.

Cada nodo de octree tiene ocho niños o sin niños. El nodo raíz describe un cuadro de límite cúbico que encapsula todos los puntos. Cada nivel del árbol, está subdividido por un factor de 2 que se traduce en un aumento de la resolución del voxel. La aplicación proporciona a octree eficientes rutinas de búsqueda a los vecinos más cercanos, como “Neighbors within Voxel Search”,

"K Nearest Neighbor Search" y "Neighbors within Radius Search". Se ajusta automáticamente su dimensión para el conjunto de datos.

Un conjunto de clases de nodo hoja proporcionan funcionalidades adicionales, como la "ocupación" y "densidad de puntos por cada voxel". Funciones para la serialización y deserialización permiten codificar de manera eficiente la estructura octree en un formato binario. Además, una aplicación de memoria reduce la asignación de memoria y desasignación de operaciones en escenarios en los que octree necesita ser creado a una alta velocidad.

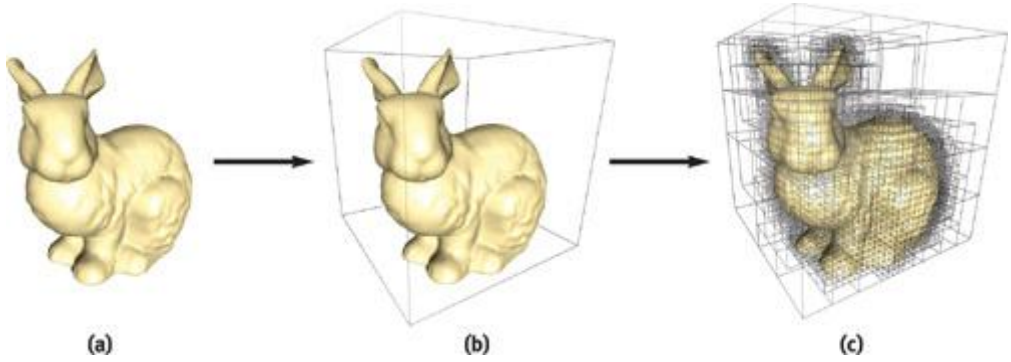


Figura 16. Octree de PCL. Disponible en:
http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter37.html

Segmentation.

La librería de la segmentación contiene algoritmos para la segmentación de una nube de puntos en distintos grupos. Estos algoritmos son los más adecuados para el procesamiento de una nube de puntos que se compone de una serie de regiones aisladas espacialmente. En tales casos, la agrupación se utiliza a menudo para romper la nube en sus partes constituyentes, que luego pueden ser procesadas de forma independiente.

Sample Consensus.

La librería sample_consensus tiene métodos SAmple Consensus (SAC) como RANSAC y modelos como planos y cilindros. Estos se pueden combinar libremente con el fin de detectar modelos específicos y sus parámetros en las nubes de puntos.

Algunos de los modelos implementados en esta librería son: líneas, planos, cilindros y esferas. Detecciones de planos se aplican a menudo a la tarea de detectar las superficies interiores comunes, como las paredes, los pisos y superficies de las mesas.

Otros modelos pueden ser utilizados para detectar y segmentar objetos con estructuras geométricas comunes (por ejemplo, ajustar un modelo de cilindro para una taza).

Surface.

La librería Surface se ocupa de la reconstrucción de las superficies originales de las exploraciones 3D. Dependiendo de la tarea en cuestión, esto puede ser, por ejemplo, crear la superficie, una representación en malla o una superficie alisada y muestreada con normales. Alisado y muestreo pueden ser importantes si la nube no es clara, o si se compone de varios análisis que no están alineados perfectamente.

La complejidad de la estimación de la superficie se puede ajustar, y las normales se pueden estimar en la misma etapa, si es necesario. Mallado es la manera general de crear una superficie de puntos, y en la actualidad hay dos algoritmos: una triangulación muy rápida de los puntos originales y un mallado más lento que suavizada y también rellena los orificios.

La creación de una superficie envolvente convexa o cóncava es útil por ejemplo cuando hay una necesidad de representación de la superficie o cuando se tienen que extraer los límites.

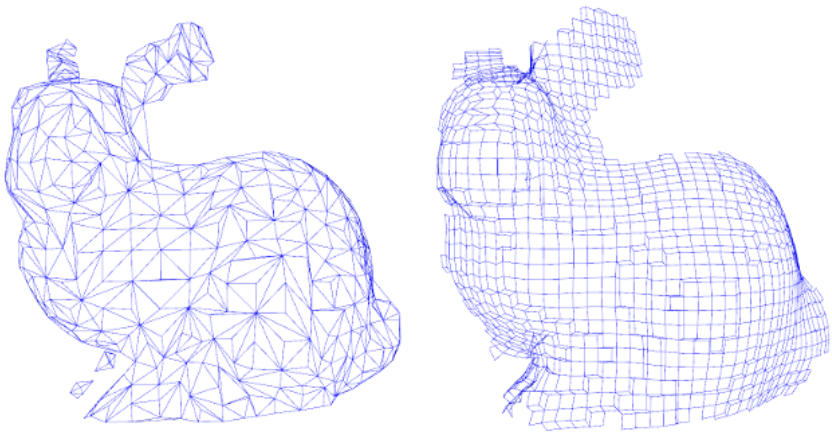


Figura 17. Surface de PCL. Disponible en:
http://docs.pointclouds.org/trunk/group__surface.html

Range Image.

La librería range_image contiene dos clases para representar y trabajar con imágenes de rango. Una imagen de rango (o mapa de profundidad) es una imagen cuyos valores de píxeles representan una distancia o profundidad desde el origen del sensor.

Las imágenes de rango son una representación 3D común y suelen generarse en estéreo o cámaras de tiempo de vuelo.

Con el conocimiento de los parámetros de calibración intrínsecos de la cámara, una imagen de rango se puede convertir en una nube de puntos.

IO.

La librería IO contiene clases y funciones para leer y escribir datos de los archivos PCD, así como la captura de nubes de puntos a partir de una variedad de dispositivos de detección.

Visualization.

La librería de visualización se construyó para permitir la rápida creación de prototipos y la visualización de los algoritmos que operan los datos de las nubes de puntos 3D.

Al igual que las rutinas highgui de OpenCV para la visualización de imágenes en 2D y para la elaboración de formas 2D básicas en la pantalla, la librería ofrece:

a) métodos de procesamiento y la configuración de las propiedades visuales (colores, tamaños de punto, opacidad, etc) para cualquier punto nD conjuntos de datos de la nube en pcl: : Formato PointCloud <T>.

b) métodos para dibujar formas básicas en 3D en la pantalla (por ejemplo, cilindros, esferas, líneas, polígonos, etc) procedentes de conjuntos de puntos o de las ecuaciones paramétricas.

c) un módulo de visualización de histograma (PCLHistogramVisualizer) para gráficos 2D;

d) una multitud de Geometría y color de controladores para pcl :: PointCloud <T> conjuntos de datos.

e) un módulo de visualización RangeImage pcl ::

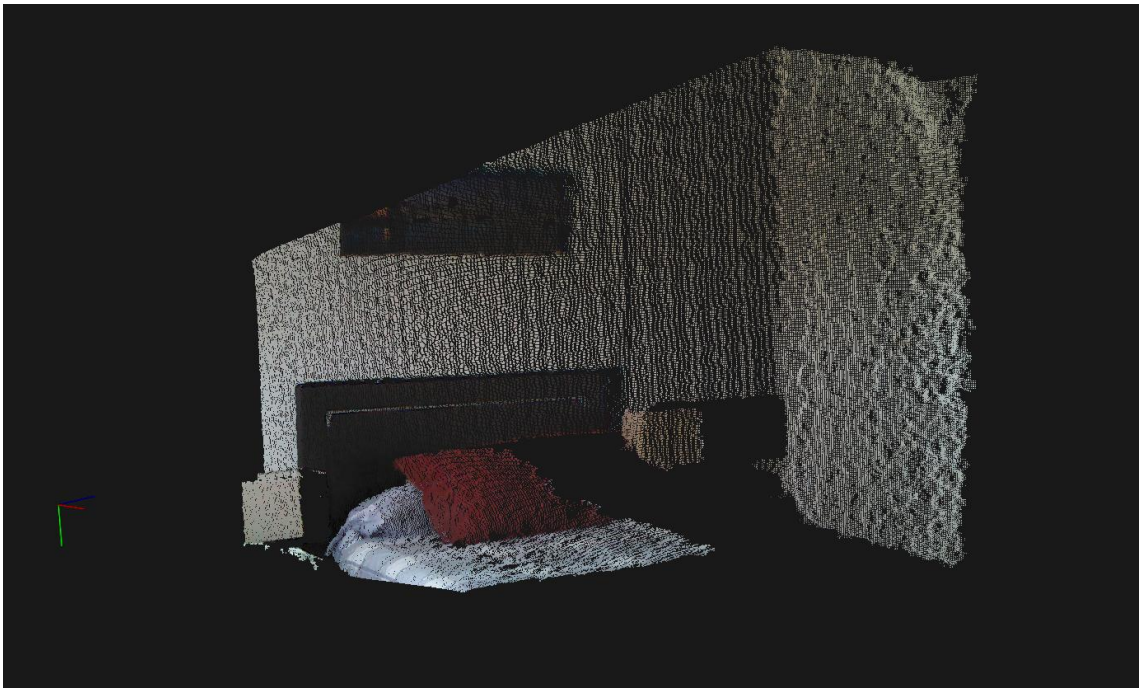


Figura 18. Visualizador de PCL.

Common.

La librería Common contiene las estructuras de datos comunes y los métodos utilizados por la mayoría de las librerías de PCL.

Las estructuras de datos principales incluyen la clase PointCloud y una multitud de tipos de puntos que se utilizan para representar puntos, superficies normales, los valores de color RGB, función de descriptores, etc. También contiene numerosas funciones computo distancia/normales, medias y covarianzas, conversiones angulares , transformaciones geométricas, y más.

La librería PCL está bajo los términos de la licencia BSD. Lo que quiere decir que es totalmente libre para su uso en investigación y su comercialización.

Su página oficial es http://pointclouds.org/

La página oficial presenta un buen entorno para el usuario. Con multitud de tutoriales dedicados a cada una de sus librerías. Además este software se encuentra en continua evolución y actualización, gracias a la gran cantidad de desarrolladores e investigadores que crean y establecen nuevos códigos y algoritmos.

Es un sistema multiplataforma compatible con los sistemas operativos de Linux, Microsoft Windows y Apple Mac OS, además se puede implementar en Qt Creator, Visual Studio, CodeBlocks y otros.

Para adquirir los datos del sensor Primesense, PCL utiliza internamente OpenNI y para la visualización de la nube de puntos utiliza VTK con algunas modificaciones.

3.5. Visual Studio 2010.

Microsoft Visual Studio es un entorno de desarrollo integrado (IDE), diseñado para el sistema operativo de Windows. Con el que podemos programar en diferentes lenguajes de programación: C++, C#, J#, NET, Basic…

Existen versiones gratuitas de Visual Studio que Microsoft proporciona desde su página oficial. Estos productos se denominan Express y son ediciones básicas en las que podemos encontrar cada lenguaje de programación por separado.

En la versión 2010 se han incorporado una serie de novedades, como las herramientas para el desarrollo de aplicaciones para Windows 7 y también uso de varios monitores al mismo tiempo. Además, también se pueden realizar aplicaciones para Windows Phone 7 entre otros.

Las diferentes ediciones de Visual Studio 2010 que podemos encontrar en el mercado son:

- Visual Studio 2010 Ultimate
- Visual Studio 2010 Premium
- Visual Studio 2010 Professional
- Visual Studio Team Foundation Server 2010
- Visual Studio Test Professional 2010
- Visual Studio Team Explorer Everywhere 2010

Para la realización de este proyecto, se ha utilizado la edición Visual Studio 2010 Professional de 32 bits, ya que Microsoft no ha emitido ninguna versión de 64 bits. Este pequeño contratiempo hace que aunque tengamos un Windows 7 de 64 bits tengamos que utilizar todas las herramientas y soportes de 32 bits para que no haya incompatibilidades.

Visual Studio 2010 Professional se puede encontrar en la página oficial de Microsoft: http://www.microsoft.com

3.6. CMake.

Esta herramienta multiplataforma se utiliza para la generación o automatización de código. El nombre de CMake proviene de la abreviatura de "cross platform make". Este software es una suite separada y de más alto nivel que el sistema make utilizado por Unix.

CMake se utiliza para controlar el proceso de compilación de software utilizando una serie de ficheros sencillos e independientes de la plataforma. Dichos ficheros se denominan CMakeLists.txt y contienen en una serie de comandos que permiten controlar el proceso de construcción.

Al contrario del GNU build system, que está restringido a plataformas Unix, CMake está diseñado para soportar la generación de los ficheros CMakeLists.txt para varios sistemas operativos, facilitando el mantenimiento y eliminando la necesidad de tener varios conjuntos de ficheros para cada plataforma.

Para nuestro caso concreto, utilizaremos un solo archivo CMakeLists.txt con los siguientes comandos:

cmake_minimum_required(VERSION 2.8 FATAL_ERROR)

*project(**NAME_PROJECT**)*

find_package(PCL 1.2 REQUIRED)

include_directories(${PCL_INCLUDE_DIRS})

link_directories(${PCL_LIBRARY_DIRS})

add_definitions(${PCL_DEFINITIONS})

*add_executable (**NAME_PROJECT NAME_CODE_FILE**.cpp)*

*target_link_libraries (**NAME_PROJECT** ${PCL_LIBRARIES})*

4. DEFINICIÓN DE DRIVERS.

Al tratarse de un controlador abierto, con Kinect se pueden utilizar diferentes drivers. En ellos se encuentran OpenNI, Kinect for Windows SDK, OpenKinect o Libfreenect.

En este proyecto se quiso comparar OpenNI y Kinect for Windows SDK debido a la gran similitud entre ambos software y finalmente se eligió OpenNI debido a que es el sistema que utiliza PCL para la adquisición de los datos de los dispositivos Primesense.

4.1. KINECT FOR WINDOWS SDK.

El 21 de febrero 2011 Microsoft Corporation anunció que lanzaría un kit no comercial Kinect de desarrollo de software (SDK) para Windows en la primavera de 2011, que fue lanzado para Windows 7 el 16 de junio de 2011 en 12 países.

El SDK incluye controladores de Windows 7 para PC compatibles para el dispositivo Kinect. Proporciona capacidades de Kinect a los desarrolladores para crear aplicaciones con C + +, C #, Visual Basic o Java con Microsoft Visual Studio 2010 e incluye las siguientes características:

1. El acceso a los flujos de bajo nivel del sensor de profundidad, sensor de cámara a color y micrófono de matriz de cuatro elementos.

2. Seguimiento esquelético: La capacidad de rastrear la imagen y esqueletizar a una o dos personas que se desplazan dentro del campo de visión de Kinect.

3. Funciones de audio avanzadas: las capacidades de procesamiento de audio incluyen supresión de ruido acústico y cancelación de eco, formación de haz para identificar la fuente de sonido actual, y la integración con la API de reconocimiento de voz de Windows.

4. Códigos de ejemplos y documentación.

En marzo de 2012, Craig Eisler, gerente general de Kinect para Windows, dijo que cerca de 350 empresas están trabajando con Microsoft en las aplicaciones personalizadas Kinect para Windows.

La versión 1.0 se lanzó en Febrero del 2012. Esta versión posee la cualidad de poder hacer comerciales aquellas aplicaciones que desarrollemos para Windows.

Las mejoras de esta nueva versión son:

- Conexión de hasta 4 sensores Kinect al mismo tiempo.
- Modificación de algunos archivos y funciones.
- Mejora de la estabilidad del conductor, recepción de audio y tiempos de computo.
- Cálculo de las imágenes tomadas por segundo.

- La resolución de la imagen RGB pasa de 1280x1024 a 1280x960 .
- Modificación y aumento de funciones para el cálculo y conversión de mapas.
- Incorporación del NearMode.
- Aparición del Motor para la inclinación.
- Etc.

En la siguiente figura se muestra la arquitectura del KINECT FOR WINDOWS SDK.

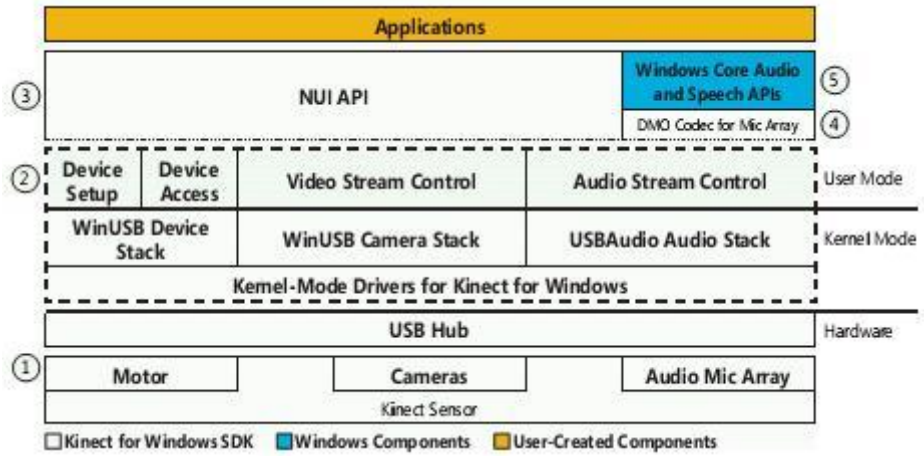


Figura 19. Arquitectura de Kinect Windows SDK. Disponible en:
http://msdn.microsoft.com/en-us/magazine/jj553517.aspx

4.2. OpenNI.

El framework desarrollado por PimeseSense, Open Natural Interaction u OpenNI es una recopilación de APIs para dispositivos que utilizan Interacción Natural. Mejorando su interoperabilidad, las aplicaciones que utilizan, el middleware que facilita el acceso y el uso de estos dispositivos, en un marco multiplataforma. OpenNI se compone de un conjunto de interfaces para la creación de aplicaciones NI.

Con Interacción Natural (NI) nos referimos a los dos sentidos principales que utiliza el ser humano para interactuar con el dispositivo, que son: la visión y la audición.

OpenNI puede ser distribuido libremente ya que se encuentra bajo la Licencia Publica General de GNU. Por lo que se puede modificar y redistribuir bajo los términos de dicha licencia.

Podemos encontrar todo lo relacionado con OpenNI en su página principal: http://www.openni.org/.

El objetivo principal de OpenNI es formar una API estándar que permite la comunicación entre:

- Los sensores encargados de la adquisición de datos del entorno; tanto de visión como auditivos.
- Los componentes del software que se encargan de analizar el audio y los datos visuales captados por los sensores.

OpenNI proporciona un conjunto de APIs para ser implementadas por los dispositivos con sensor/es, y un conjunto de APIs que son implementados por los componentes de middleware.

Al romper la dependencia entre el sensor y el middleware, la API de OpenNI permite escribir aplicaciones y portarlas sin esfuerzo adicional para operar en la parte superior de diferentes módulos middleware (Una vez escrito, lo desplegamos por todas partes).

Este framework permite grabar secuencias de imágenes a color y 3D en archivos de extensión propia (.oni). A la hora de crear este archivo se puede elegir diversos tipos de compresión.

A diferencia del SDK de Windows, OpenNI permite su utilización mediante cualquier dispositivo que posea el chip PS1080 de PrimeSense.

Estos dispositivos son:

- ASUS Xtion PRO Live
- ASUS Xtion PRO
- PrimeSense PSDK
- Microsoft Kinect

Figura 20. Dispositivos con chip PS1080 de PrimeSense. Disponible en: http://pointclouds.org/documentation/tutorials/openni_grabber.php#openni-grabber

OpenNI permite reconocer comandos de voz. También reconoce gestos de las manos predefinidos y con ambos es capaz de interpretarlos para activar y controlar los dispositivos.

Otro recurso de esta tecnología es que permite el reconocimiento y seguimiento del cuerpo humano, lo que en Ingles se reconoce como tracking.

OpenNI posee una serie de módulos en su interfaz para los dispositivos físicos y los componentes de middleware. Se utilizan para adquirir y procesar los datos sensoriales.

Los módulos soportados por OpenNI son los siguientes:

- Sensores 3D
- Dispositivos de audio
- Cámaras RGB
- Cámaras de infrarrojos

Los componentes middleware que se pueden utilizar para analizar son:

- Detección, seguimiento y esqueletización del cuerpo.
- Detección y seguimiento de la mano.
- Detección de gestos predefinidos.
- Análisis del entorno.

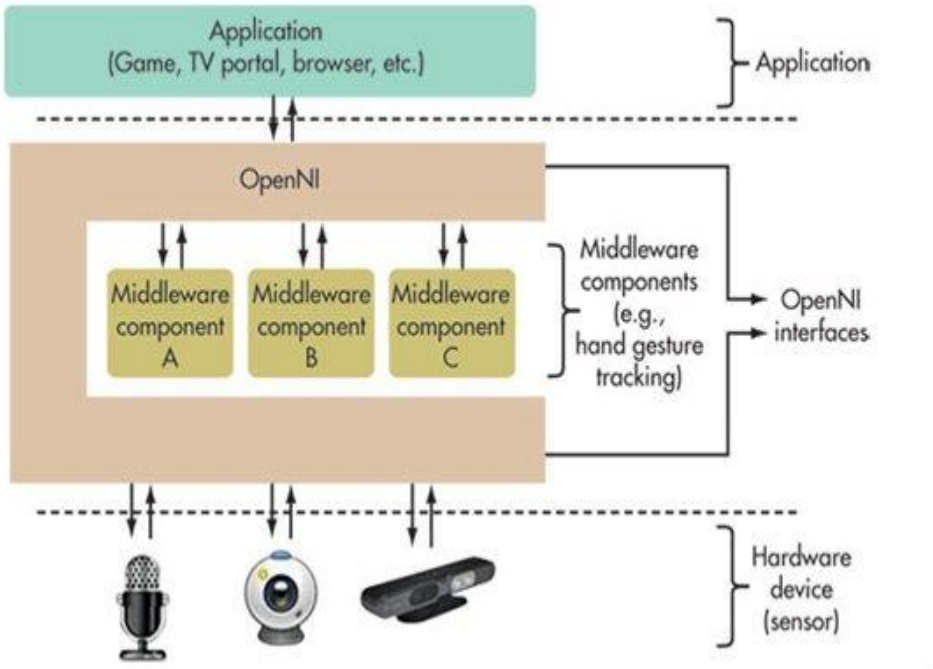


Figura 21. Arquitectura de OpenNI. Disponible en:
http://social.technet.microsoft.com/wiki/contents/articles/6370.working-of-microsoft-s-primesense-technology-based-kinect-an-elaboration.aspx

5. COMPARACIÓN ENTRE SDK DE WINDOWS Y OPENNI.

Para saber que framework debemos utilizar para la resolución del proyecto, en este apartado se realizara una comparación entre ellas.

Tipo de licencia:

En primer lugar, encontramos una gran diferencia en cuanto a las licencias de uso de cada librería. El SDK de Windows utiliza una licencia propietaria de Microsoft, la cual nos impone sus propias condiciones para el uso de Kinect bajo este framework. Por otro lado, OpenNI se encuentra bajo la licencia pública general de GNU, lo que nos permite modificar y redistribuir bajo los términos de dicha licencia.

Instalación.

En cuanto al modo de instalación de cada librería, se puede encontrar algunas diferencias notables respecto a la dificultad de este proceso. La librería de Microsoft, presenta una instalación práctica y sencilla, solo debemos acceder a su página oficial y seleccionar el ejecutable de la versión que deseemos.

Mientras que la instalación del SDK de Windows, es sencilla, la instalación de OpenNI es más confusa, ya que hay que tener bastante claro los archivos que debemos instalar:

- OpenNI Binaries
- OpenNI Hardware
- OpenNI Compliant
- SensorKinect-Bin-Win32

Sin embargo, existe un paquete de instalación completo desarrollado por ZigFu para OpenNI. Es un ejecutable que se encarga de instalar todos los archivos necesarios para la correcta instalación y funcionamiento de OpenNI en nuestro computador y es el que se ha utilizado en este proyecto.

El paquete de instalación de ZigFu se puede encontrar en: http://zigfu.com/en/downloads/browserplugin/

Lenguajes de programación.

Los lenguajes que podemos utilizar para programar con el SDK son: C, C++, C#, VB y Java.

Mientras que para OpenNI solamente podemos utilizar como lenguajes de programación C y C++.

Plataformas de desarrollo.

En este punto, también se puede apreciar una diferencia considerable. Para poder usar Kinect Windows SDK hace falta, obligatoriamente, trabajar con Windows 7 nativo y usar la versión de Visual Studio 2010, por otro lado con OpenNI se puede usar diferentes entornos de desarrollo y puede utilizarse en diferentes sistemas operativos.

Dispositivos.

El SDK de Windows, solamente sirve con su propio sensor Kinect de Microsoft.

Sin embargo OpenNI se puede utilizar con todos los sensores que tengan el chip PS1080 de PrimeSense.

Estos dispositivos son:

- ASUS Xtion PRO Live
- ASUS Xtion PRO
- PrimeSense PSDK
- Microsoft Kinect

Utilidades.

SDK puede utilizar todas las funciones de Kinect:

- Emisor IR.
- Dispositivos de audio.
- Cámara RGB.
- Cámara de infrarroja.
- Motor.
- Near Mode.

Mientras que OpenNI solo puede usar las funciones de Kinect:

- Emisor IR.
- Dispositivos de audio.
- Cámara RGB.
- Cámara de infrarroja.

Por otro lado OpneNI es capaz de realizar grabación de una secuencia que SDK de Windows no tiene. Y otra grandísima diferencia es que las Librerías PCL, trabajan internamente con OpenNI.

ELECCIÓN.

La elección final de un framework u otro, dependerá de las preferencias y necesidades de cada usuario a la hora de realizar su proyecto. Y dado que en nuestro caso particular, necesitamos trabajar con PCL, es un sistema multiplataforma y puede realizar grabaciones de secuencias con Kinect. Por todo ello, se ha elegido OpenNI para la realización de este proyecto.

6. PLANTEAMIENTO DEL PROBLEMA.

Como ya hemos mencionado anteriormente, nuestro objetivo es obtener la esqueletización 3D de los brazos del conductor de un vehículo para poder percibir sus movimientos dentro del mismo.

Para conseguir nuestro objetivo, es necesario instalar y fijar nuestro sensor Kinect en el techo del interior del vehículo. Deberemos tener en cuenta cual será mejor posición para realizarlo.

Después de realizar varias pruebas llegamos a la conclusión de que la mejor posición es justo encima del reposacabezas del conductor (ya que si la situamos en un lateral, pueden aparecer sobras de un brazo sobre el otro), con una cierta inclinación para el correcto encuadre de la imagen. En dicho encuadre deben aparecer tanto los brazos del conductor, como la totalidad del volante, Radio/CD, palanca de cambios y el resto de controles que suele utilizar el conductor cuando maneja el vehículo.

Se debe tener en cuenta que como el asiento del conductor se puede desplazar de delante hacia atrás, la cámara debería poder realizar el mismo desplazamiento que el asiento, otra posibilidad sería que el asiento se encuentre fijo en todo momento, con el objetivo de que se mantenga el correcto encuadre de la imagen.

Otro hecho a tener en cuenta son los rayos infrarrojos presentes en la luz solar, debido a que nuestro sensor es muy sensible a estos rayos y nuestra aplicación se va a utilizar en el exterior. En principio no debería haber problema, ya que los cristales de nuestro vehículo filtran bastante bien estos rayos, pero si en algún momento se bajan las ventanillas es muy probable que nuestro programa presente errores.

7. EXPERIMENTACIÓN.

Para la realización de nuestra aplicación se han seguido los siguientes pasos, los cuales procuran optimizar el reconocimiento y la esqueletización de los brazos del conductor.

- Adquisición de los datos a través de nuestro sensor Kinect.
- Capturar una nube de puntos Base, sin el conductor.
- Detectar al conductor, comparando nuestra imagen base con el resto de imágenes capturadas por nuestro sensor.
- Segmentar los brazos del resto de partes del conductor.
- Obtener el "esqueleto" de los brazos del conductor.

A continuación se detallan cada uno de los pasos anteriormente mencionados:

7.1 Adquisición de datos.

En primer lugar, nuestro sistema adquiere los datos por parte del sensor Kinect, de esta función se encarga PCL, utilizando internamente la herramienta OpenNI.

OpenNI obtiene los datos de profundidad y RGB por separado y es PCL, con VTK, los que se encargar de ordenarlos de tal forma que los datos se puedan visualizar conjuntamente como nube de puntos y trabajar de esta manera con las bibliotecas de PCL.

```
//Con las siguientes líneas de código se captura las nubes de
puntos en tiempo real del sensor kinect, hasta que se cierra la
ventana del visualizador.
pcl::Grabber* interface = new pcl::OpenNIGrabber();
boost::function<void (const
pcl::PointCloud<pcl::PointXYZRGBA>::ConstPtr&)> f = boost::bind
(&SkeletonViewer::cloud_cb_, this, _1);
grabber->registerCallback (f);
grabber->start ();
while (!viewer.wasStopped())
 {
     boost::this_thread::sleep (boost::posix_time::seconds (1));
 }
grabber->stop ();
```

Otra forma de obtención de los datos es mediante una grabación realizada con OpenNI en un archivo de formato .oni. Con este método podemos trabajar

sobre esta grabación sin necesidad de realizar pruebas constantemente en el vehículo y sin la necesidad de tener conectado el sensor Kinect. Lo que le convierte en un método bastante más cómodo para programar nuestra aplicación y una vez finalizada, sólo hay que pasar al método anterior para que pueda funcionar con la obtención de datos en tiempo real.

Un inconveniente de este método, es que sólo se puede ver una vez la grabación por cada ejecución del programa.

```
//Con las siguientes líneas de código se captura las nubes de
puntos en un archivo de grabación .oni. Hasta que cerramos la
ventana del visualizador.
pcl::ONIGrabber* grabber = new pcl::ONIGrabber("captura.oni",
false, true);
boost::function<void (const
pcl::PointCloud<pcl::PointXYZRGBA>::ConstPtr&)> f = boost::bind
(&SkeletonViewer::cloud_cb_, this, _1);
grabber->registerCallback (f);
grabber->start ();
while (!viewer.wasStopped())
{
         boost::this_thread::sleep (boost::posix_time::seconds
         (1));
 }
grabber->stop ();
```

7.2 Captura de Nube de Puntos Base.

Una vez que somos capaces de obtener los datos del sensor. El primer paso es capturar una nube de puntos “base”, en la que se vea el puesto de conducción del vehículo; pero sin el conductor.

Para ello se visualizan los datos captados por el sensor Kinect en el momento en el que no se encuentre sentado el conductor, presionaremos la letra ‘s’ para guardar ese frame en un archivo de almacenamiento .pcd (Point Cloud Data). De esta forma tendremos guardados todos los puntos pertenecientes al vehículo.

```
stringstream stream;
stream << "inputCloud" << save_one << ".pcd"; // Creamos una
cadena de caracteres con el nombre de nuestro archivo .pcd
string filename = stream.str();
if (pcl::io::savePCDFile(filename, *cloud, true) == 0) //
Guardamos el frame y comprobamos si se ha guardado correctamente
cout << "Saved " << filename << "." << endl;
else
```

```
PCL_ERROR("Problem saving %s.\n", filename.c_str());
```

En la siguiente figura se observa la nube de puntos del puesto de conducción vacía que vamos a utilizar como nube de puntos "Base".

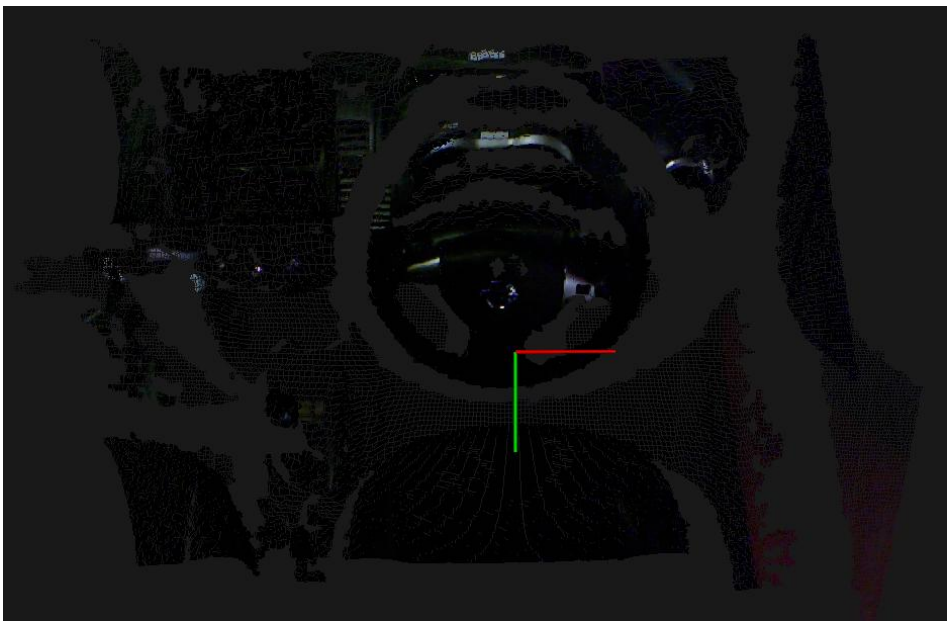


Figura 22. Captura Nube de Puntos Base.

7.3 Detección del Conductor.

El siguiente paso es diferenciar al conductor del resto del vehículo. Para ello se toma la imagen base y se carga en el método iterativo de búsqueda octree.

```
//En estas lineas de código se lee la nube de puntos base,
almacenada en el archivo pcd y se carga en la nube cloudBase.
pcl::PointCloud<pcl::PointXYZRGBA>::Ptr cloudBase (new
pcl::PointCloud<pcl::PointXYZRGBA>);
if (pcl::io::loadPCDFile<pcl::PointXYZRGBA> ("inputCloud1.pcd",
*cloudBase) == -1) //load the file
{
      PCL_ERROR ("Couldn't read file inputCloud1.pcd \n");

}
//Posteriormente se añade la nube cloudBase a octree.
std::cerr << cloudBase->points.size() << " -- ";
// assign point cloud to octree
octree->setInputCloud (cloudBase);
// add points from cloud to octree
```

```
octree->addPointsFromInputCloud ();
// switch buffers - reset tree
octree->switchBuffers ();
```

Posteriormente se añaden a octree cada uno de los frames que va capturando nuestro sensor en tiempo real y se obtiene las diferencias entre la nube de puntos base y la nueva captada por Kinect.

```
//Una vez cargada la nube "Base" se carga la que se desea
comparar "cloud".
std::cerr << cloud->points.size() << " -- ";
// assign point cloud to octree
octree->setInputCloud (cloud)
// add points from cloud to octree
octree->addPointsFromInputCloud ();
std::cerr << octree->getLeafCount() << " -- ";
```

Cada uno de los puntos diferentes entre ambas nubes de puntos, serán los correspondientes al conductor de nuestro vehículo.

```
boost::shared_ptr<std::vector<int> > newPointIdxVector (new
std::vector<int>);
// Se cargan los puntos diferenciados en el vector
newPointIdxVector.
// get a vector of new points, which did not exist in previous
buffer
octree->getPointIndicesFromNewVoxels (*newPointIdxVector,
noise_filter_);
std::cerr << newPointIdxVector->size() << std::endl;
pcl::PointCloud<pcl::PointXYZRGBA>::Ptr filtered_cloud;
```

Para poder apreciar esta detección del conductor. Se ha dotado a nuestra aplicación de una opción de visualización del conductor. En la cual marcamos de color rojo todos los puntos de la nube que se corresponden al conductor.

```
pcl::PointCloud<pcl::PointXYZRGBA>::Ptr filtered_cloud;
// En este caso se muestra toda la nube de puntos actual, con
los puntos diferenciados de la imagen actual, marcados de color
rojo.
filtered_cloud = (pcl::PointCloud<pcl::PointXYZRGBA>::Ptr) new
pcl::PointCloud<pcl::PointXYZRGBA> (*cloud);
filtered_cloud->points.reserve(newPointIdxVector->size());
for (std::vector<int>::iterator it = newPointIdxVector->begin
(); it != newPointIdxVector->end (); it++)
filtered_cloud->points[*it].rgb = 255<<16;
```

En la siguiente figura se observa la nube de puntos con el conductor detectado y representado de color rojo.

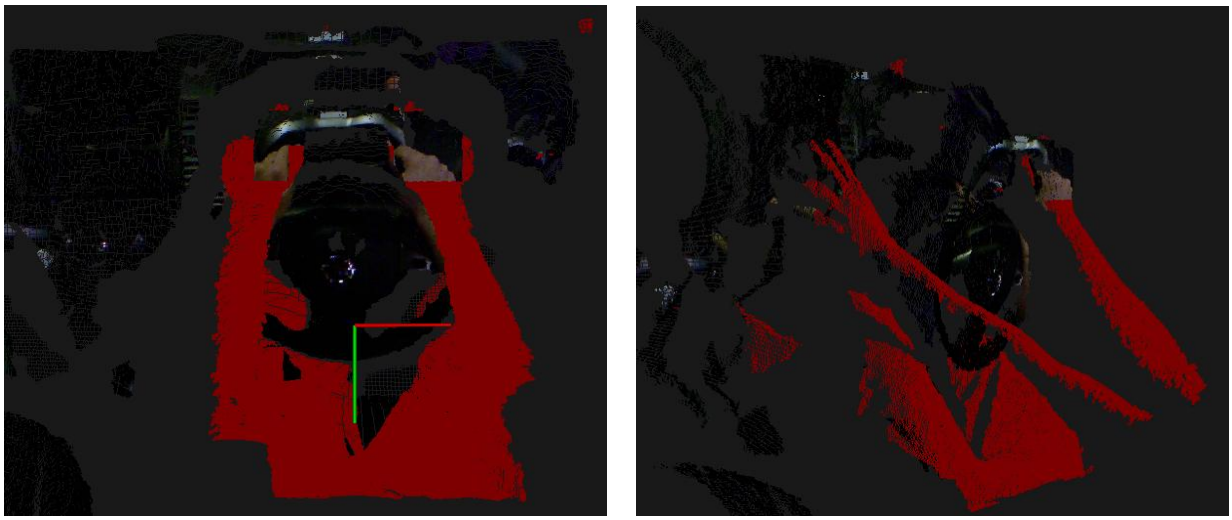


Figura 23. Detección del conductor.

7.4 Segmentación de los Brazos.

Por el encuadre del la imagen, las partes visibles del conductor son los brazos y las piernas. Una vez que se ha detectado al conductor y para cumplir con nuestro objetivo de analizar su actividad en el puesto de conducción, se deben separar los brazos de las piernas.

Para ello y en primer lugar, se usa un filtro VoxelGrid para reducir a 1 punto por centímetro la nube de puntos. Esta reducción de información permitirá realizar el resto de operaciones mucho más rápido.

```
// Create the filtering object: downsample the dataset using a
leaf size of 1cm
pcl::VoxelGrid<pcl::PointXYZRGBA> vg; //Declaramos el filtro con
el tipo de nube utilizada.
pcl::PointCloud<pcl::PointXYZRGBA>::Ptr cloud_filtered (new
pcl::PointCloud<pcl::PointXYZRGBA>);// Nube donde se guarda el
resultado
vg.setInputCloud (filtered_cloud); // Nube que contiene los
datos del conductor.
vg.setLeafSize (0.01f, 0.01f, 0.01f);// Tamaño de los Voxels
vg.filter (*cloud_filtered); // Nube filtrada.
std::cout << "PointCloud after filtering has: " <<
cloud_filtered->points.size ()  << " data points." << std::endl;
```

Una vez se ha reducido el tamaño de la nube de puntos con la que vamos a trabajar, se realiza una búsqueda de los diferentes objetos con kdtree para separarlos en Clusters atendiendo a que los puntos componentes de dicho cluster estén a una distancia Euclidea determinada.

```
pcl::EuclideanClusterExtraction<pcl::PointXYZRGBA> ec;// Método
de segmentación de objetos.
ec.setClusterTolerance (0.02); // Distancia máxima entre puntos
para ser del mismo objeto.
ec.setMinClusterSize (100); // Número de puntos mínimos del
cluster para ser objeto.
ec.setMaxClusterSize (30000); //Número de puntos máximos del
cluster para ser objeto.
ec.setSearchMethod (tree); //método de búsqueda elegido kdtree
ec.setInputCloud (cloud_filtered);
ec.extract (cluster_indices); //extracción de los índices de los
objetos
```

Para finalizar, se calculan los centroides de cada objeto encontrado en el paso anterior. Posteriormente se filtran los objetos por la profundidad de sus centroides y así se consigue separar los brazos de las piernas. Para separar los brazos entre sí, sólo hay que filtrar en el eje de las X y así se consiguen segmentar los brazos en derecho e izquierdo.

En este caso también se ha dotado a la aplicación de la opción de visualizar los dos brazos del conductor, totalmente segmentados.

```
// Una vez que tenemos los objetos diferenciados, calculamos su
centroide y descartamos por profundidad.
int j = 0;
for (std::vector<pcl::PointIndices>::const_iterator it =
cluster_indices.begin (); it != cluster_indices.end (); ++it)
{
      pcl::PointCloud<pcl::PointXYZRGBA>::Ptr cloud_cluster (new
      pcl::PointCloud<pcl::PointXYZRGBA>);
      for (std::vector<int>::const_iterator pit = it-
      >indices.begin (); pit != it->indices.end (); pit++)
      cloud_cluster->points.push_back (cloud_filtered-
      >points[*pit]);
      cloud_cluster->width = cloud_cluster->points.size ();
      cloud_cluster->height = 1;
      cloud_cluster->is_dense = true;
      compute3DCentroid (*cloud_cluster, centroid);
      //Si el objeto pasa el filtro de profundidad y cumple que
      se encuentra en una posición menor a 0 en el eje de las X,
      quiere decir que se trata del brazo/izquierdo
```

```
        if(0.85>centroid[2] && 0.0>centroid[0])
        *cloudBrazo1+=*cloud_cluster;
        //Si el objeto pasa el filtro de profundidad y cumple que
        se encuentra en una posición mayor a 0 en el eje de las X,
        quiere decir que se trata del brazo derecho
        if(0.7>centroid[2] && 0.0<centroid[0])
        *cloudBrazo2+=*cloud_cluster;
        j++;
}
```

En la siguiente figura se observa la nube de puntos con los brazos totalmente segmentados (sin las piernas del conductor ni partes del vehículo).

Figura 24. Segmentación de los Brazos.

7.4 Esqueletización de los Brazos.

El último paso es obtener la esqueletización y las articulaciones de los brazos. Para ello se divide cada brazo en 15 trozos iguales y se calcula el centroide de cada parte.

```
for(int i=0; i<=numero_centroides; i++)
{
        pass2.setInputCloud (cloudBrazo1); // Nube que contiene el
        brazo 1.
        pass2.setFilterFieldName ("y"); // Dimensión en la que se
        filtra.
        pass2.setFilterLimits (inicio1, limitesup1); // Intervalo
        del filtrado
        //pass2.setFilterLimitsNegative (true);
        pass2.filter (cloudaux1[i]); // Nube donde se guarda el
        trozo de brazo.
```

```
      //Se calcula el centroide de cada parte
      compute3DCentroid (cloudaux1[i], centroid1[i]); // Se
      calcula el centroide del trozo de brazo.
      inicio1=limitesup1;
      limitesup1+=final1;
}
```

Se introducen todos los centroides obtenidos en una nube de puntos.

```
cloud_centroid1->width =   cloudBrazo1->width;
cloud_centroid1->height =cloudBrazo1->height;
cloud_centroid1->points.resize (cloud_centroid1->width *
cloud_centroid1->height);
for(int i=0; i<=numero_centroides;i++)
{
      loud_centroid1->points[i].x = centroid1[i][0];
      cloud_centroid1->points[i].y = centroid1[i][1];
      cloud_centroid1->points[i].z = centroid1[i][2];
}
```

Y por último, según de la cantidad de puntos o tamaño del brazo que captura el sensor, se llama a las funciones que dibujan el esqueleto de los brazos uniendo los centroides de cada sub-trozo y añade una esfera en representación de las articulaciones (muñecas y codos).

```
/*Los siguientes if separan los 3 casos de posición del brazo
que podemos encontrar;
1. Tenemos muy poco brazo dentro de la imagen (la mano, muñeca y
muy poco antebrazo).
2. Tenemos bastante trozo de brazo pero sin el codo.
3. Tenemos el brazo prácticamente completo, incluyendo el codo.
*/
if(525>cloudBrazo1->width)
{
      viewer.runOnVisualizationThreadOnce(viewer_remove1);
      //Llama a la función que borra todo el brazo que estaba
      dibujado del frame anterior.
      viewer.runOnVisualizationThreadOnce(viewerBrazo1_pequeño);
      //Llama a la función que dibuja el trozo de brazo
      correspondiente.
}
if(525<cloudBrazo1->width)
{
      viewer.runOnVisualizationThreadOnce(viewer_remove1);
      //Llama a la función que borra todo el brazo que estaba
      dibujado del frame anterior.
```

```
      viewer.runOnVisualizationThreadOnce(viewerBrazo1); //Llama
      a la función que dibuja el trozo de brazo correspondiente.
}
if(600<cloudBrazo1->width)
      viewer.runOnVisualizationThreadOnce(viewerCodo1); //Llama a
      la función que dibuja el trozo de brazo correspondiente.

// Fución que se encarga de dibujar el esqueleto del brazo
izquierdo cuando solo se ve una pequeña parte de el.
void viewerBrazo1_pequeño (pcl::visualization::PCLVisualizer&
viewer)
{
      // Con este bucle dibujamos el trozo de esqueleto
      correspondiente.
      for(int i=1; i<=(numero_centroides-12);i++)
      {
            std::stringstream ss ("line5");
            ss << i;
            viewer.addLine (cloud_centroid1->points[i],
            cloud_centroid1->points[i+1], 0,0.5, 0, ss.str ());

      }
      // Se añade una esfera para representar la articulación de
      la muñeca.
      viewer.addSphere (cloud_centroid1->points[1], 0.03, 0.0,
      0.5, 0.0, "sphere1");
}
// Fución que se encarga de dibujar el esqueleto del brazo
izquierdo cuando solo se ve bastante parte del antebrazo, sin el
codo.
void viewerBrazo1 (pcl::visualization::PCLVisualizer& viewer)
{
      // Con este bucle dibujamos el trozo de esqueleto
      correspondiente.
      for(int i=2; i<=(numero_centroides-4);i++)
      {
            std::stringstream ss ("line1");
            ss << i;
            viewer.addLine (cloud_centroid1->points[i],
            cloud_centroid1->points[i+1], 0,0.5, 0, ss.str ());
      }
      // Se añade una esfera para representar la articulación de
      la muñeca.
      viewer.addSphere (cloud_centroid1->points[4], 0.03, 0.0,
      0.5, 0.0, "sphere1");
}
// Fución que se encarga de dibujar el resto del esqueleto del
brazo izquierdo y el codo.
```

```
void viewerCodo1 (pcl::visualization::PCLVisualizer& viewer)
{
      // Con este bucle dibujamos el trozo de esqueleto
      correspondiente.
      for(int i=10; i<=(numero_centroides-2);i++)
      {
            std::stringstream ss ("line1");
            ss << i;
            viewer.addLine (cloud_centroid1->points[i],
            cloud_centroid1->points[i+1], 0,0.5, 0, ss.str ());
      }
      // Se añade una esfera para representar la articulación del
      codo.
      viewer.addSphere (cloud_centroid1->points[12], 0.03, 0.0,
      0.5, 0.0, "sphere2");
}
```

Se realizan los mismos pasos para el segundo brazo, regulando algunos parámetros en función de las posiciones en las que vaya a encontrarse.

En la siguiente figura se observa la nube de puntos final, con la esqueletización de los brazos.

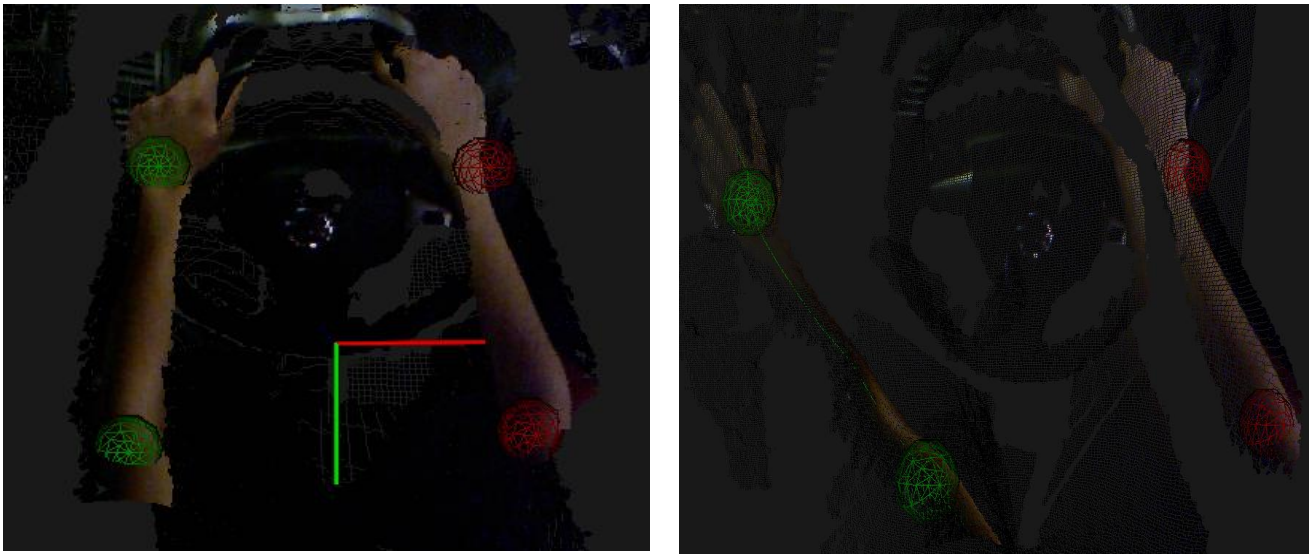
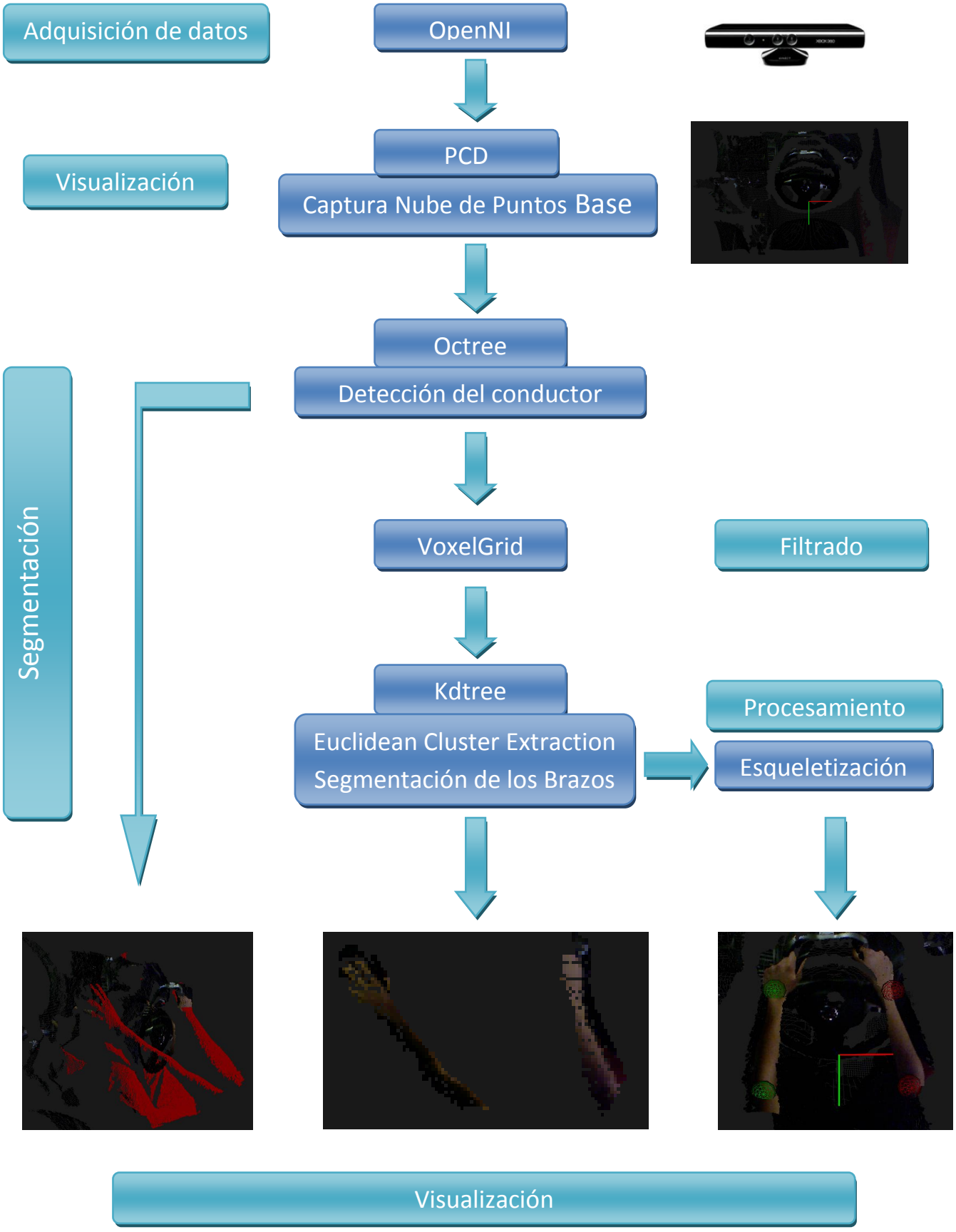


Figura 25. Esqueletización y seguimiento de los Brazos.

EL RESULTADO FINAL ESTÁ DISPONIBLE EN:

http://www.youtube.com/watch?v=e3WqOmmr8hs

ESQUEMA DEL PROCESO



8. CONCLUSIONES.

Durante la realización de este Trabajo Fin de Grado se ha conseguido una aplicación capaz de:

- Adquirir y procesar los datos de profundidad y RGB del sensor Microsoft Kinect tanto en tiempo real como los grabados en un archivo de almacenamiento en formato .oni.

- Capturar una nube de puntos Base, del puesto de conducción, y almacenarla en un archivo de almacenamiento PCD (*Point Cloud Data*), para posteriormente compararla con cada frame captado por nuestro sensor.

  Al utilizar este método se consigue no tener que estar capturando la nube de puntos Base cada vez que se inicia el programa, con hacerlo una vez es suficiente.

- Detectar al conductor del vehículo entre el conjunto de datos del entorno, comparando la imagen base guardada con el resto de frames capturados por nuestro sensor Kinect en tiempo real, mediante el método iterativo anteriormente explicado. Pudiendo visualizar el resultado, con los puntos pertenecientes al conductor coloreados de rojo.

- Filtrar la nube de puntos resultante sin perder información relevante, para obtener una mayor velocidad de procesamiento.

- Segmentar los brazos de las piernas del conductor, filtrando sus centroides por profundidad y filtrando en el eje de las X para diferenciar entre un brazo y otro. Pudiendo visualizar el resultado.

- Obtener la esqueletización de ambos brazos. Troceando la nube de puntos de cada brazo y calculando los centroides de dichos trozos para posteriormente unirlos y formar así el esqueleto. Colocando una esfera en representación de cada articulación (codos y muñecas). Pudiendo visualizar el resultado final y consiguiendo el objetivo final de percibir la actividad del conductor.

8. CONCLUSIONS.

During the realization of the Grade final work, we have achieved an application able to:

- Acquire and process the deep data and the RGB of the Microsoft Kinect sensor in real time as well as in the recorded in a storage archive named .oni.

- Capture a Base point cloud of the driver seat and store it in a storage archive PCD (Point Cloud Data) for compare it with each frame captured by our sensor.

  When this method is used, it is not necessary capture the Base point clouds every time that we start the program.

- Detect the vehicle driver between the data groups of the environment; the saved base image is compared with the rest of the captured frames by our sensor Kinect in real time through an iterative method that has been explained before. Finally, the result can be visualized with the driver points colored of red.

- Filter the resulting point cloud without losing the relevant information in order to obtain a higher processing speed.

- Segment the arms from the legs of the driver through filtering the centroids by depth and through X axis to differentiate an arm from the other. Finally, it can be visualized the result.

- Obtain the skeletonization of both arms through cutting up the point clouds of each arm and calculating the centroids of each piece. Then we join them in order to form the skeleton. A sphere is put in the representation of each articulation (elbow and wrist). Finally, it can be visualized the result achieving the final objective of perceiving the driver activity.

9. POSIBLES TRABAJOS FUTUROS.

En este punto se expondrán algunos desarrollos que se pueden implementar para aumentar las funcionalidades de nuestra aplicación:

- Uno de los trabajos futuros que se deberían llevar a cabo, sería la optimización del código para reducir su tiempo de ejecución y que permita poder trabajar con la aplicación en unas condiciones que se asemejen, el máximo posible, a las de tiempo real. Para ello, se podría hacer uso de los recursos de la GPU (*Graphics Processing Unit*) e intentar utilizar un entorno de desarrollo que posea una versión de 64 bits, para poder emplear todas las herramientas utilizadas en esta versión y aprovechar su alta velocidad de cómputo.

- Otro posible trabajo futuro podría ser el análisis de la actividad del conductor dentro del vehículo, como por ejemplo: analizar el correcto posicionamiento de las manos en el volante, si tiene una mano en la palanca de cambios, si está manejando la Radio/CD (*Compact Disc*), distraído con un Smartphone, fumando, etc.

  Estos análisis podrían tener una gran importancia a la hora de prevenir accidentes o descubrir la causa de los mismos.

10. PRESUPUESTO.

Para la realización de este proyecto se han utilizado versiones gratuitas de las siguientes herramientas: *OpenCV, PCL, Visual Studio, OpenNI, SDK Kinect for Windows, paquete Primesense y CMake*.

Además, no se considerará el coste de la CPU (*Central Process Unit*) ni el del sistema operativo en este presupuesto.

Los costes a tener en cuenta son los siguientes:

- Precio sensor Kinect: 150€

- Tiempo invertido en la realización del proyecto:

  Horas invertidas por el ingeniero: 485 horas.

  Precio/hora: 30€/h.

  Total: 14.550€

Presupuesto final: 14.700€

BIBLIOGRAFÍA.

[1] Documentation Point Cloud Library: *http://pointclouds.org/documentation/tutorials/*

[2] Reto SDK de Kinect: Desarrolla con Kinect: *MSDN España.*

[3] Community-powered support for OpenNI

[4] OpenNI User Guide.

[5] Hemant M. Kakde. *Range Searching using Kd Tree, 2005.*

[6] Point Cloud Library (PCL) Users mailing list.

[7] Plagemann, C. Ganapathi, V. Koller, D. Thrun, S. *Real-time Identification and Localization of Body Parts from Depth Images*. Artificial Intelligence Laboratory, Stanford University.

[8] Raúl Reina Molina, *Esqueletización de imágenes 3D*. Escuela Técnica Superior de Ingeniería Informática Máster en Matemática Computacional, Universidad de Sevilla.

[9] Anónimo Initial alignment of point clouds, 2011.

[10] PCL API Documentation.

Nota: Las direcciones web se encuentran disponibles a 28 de Mayo de 2013.